République Algérienne Démocratique et Populaire
Ministre de l'enseignement supérieur et de la recherche scientifique

UNIVERSITE MOHAMED KHIDER BISKRA
FACULTE DES SCIENCES EXACES, DES SCIENCES DE LA NATURE ET DE LA VIE
DEPARTEMENT D'INFORMATIQUE

N° d'ordre :…..
Série :…..

THESE

Présentée pour obtenir le grade de
DOCTORAT EN SCIENCES EN INFORMATIQUE

Extending Petri Nets for Modeling and Analysis of Reconfigurable Systems

Présentée par :

M. Kahloul Laid

Soutenue le: …/ ….. /20…

Devant le jury:

Pr. Noureddine Djeddi	Université de Biskra	Président
Dr. Allaoua Chaoui	Université Mentouri Constantine	Rapporteur
Pr. Djamel Eddine Saidouni	Université Mentouri Constantine	Examinateur
Dr. Okba Kazar	Université de Biskra	Examinateur
Dr. Mustapha Bourahla	Université de M'SILA	Examinateur
Dr. Kamel Eddine Melkmi	Université de Biskra	Examinateur
Pr. Karim Djouani	Université Paris Est (UPEC)	Invité

Dedicatee

I dedicate this work to:
My mother who pries always for me to success in my life
My wife who offers many sacrifices during the achievement of this thesis

Acknowledgments

The achievement of this thesis is due to the assistant and orientations of many persons that I have meet in these last five years. I want here to present my acknowledgments to my reporter **Dr Alloaua Chaoui**, for his orientation, his patience with me and his disposability during these years. I have to thank also all the team in the Computer Science Department, where I have done my graduate and post-graduate studies, and where I am working. My thanks also are presented to the University which gave me the chance to pass two years in the laboratory LISSI (Paris 12) during the finalization of this thesis. And so, I have to thank all the team of the LISSI laboratory in Paris 12 University: **Pr Djouani Karim** who is my co-reporter and **Pr Amirat Yacine** who is the laboratory director. I hope that they find here all my acknowledgments for all the helps that they gave me during my formation in the laboratory. I want also to present my thanks to all colleagues and friends in the laboratory: Khaled, Nadeem, Safdar, Noura, Souhil, Fouzi, Hossine, Brahim, Walid, Karim, ... with whom I have shared many days, and many events, during two years in France. Finally, I present my thanks to the members of the jury who give me the honour by accepting to evaluate this work.

Abstract

Abstract: *Petri nets are a formal and graphical tool proposed to model and to analyze behavior of concurrent systems. In its basic version, this model is defined as a fixed graph, where the behavior of the system is modeled as the marking of the graph that changes over time. This constraint makes the Petri Nets a poor tool to deal with reconfigurable systems where the structure of the system can change as its behavior, during time. Many extended Petri nets were proposed to deal with this weakness. The aim of this work is to present an extension where the structure of the graph can be highly flexible. This flexibility gives a rich model with complex behaviors, not allowed in previous extensions. The second aim is to prove that even these behaviors are so complex; they can be encoded in other models and so be analyzed.*

Keywords: *Petri Nets, Dynamic Nets, Reconfigurable Systems, Extended Petri Nets, Flexible Nets.*

Table of contents

Chapter III: Extended Petri Nets

Chapter IV: Encoding of Flexible Nets into Dynamic Nets

Conclusion

Tabled of figures

Chapter I: Mobile Computing

Chapter II: Formal Methods for Mobile Computing

Chapter III: Extended Petri Nets

Chapter IV: Encoding of Flexible Nets into Dynamic Nets

Introduction

Introduction

The development of computer science technologies and the increasing of users requirements is the most reason of the born of sophisticated solutions. Mobility with its soft and hard aspects is one of these solutions. When some disaster menaces a critical system during its execution, it seems a good idea to transfer this system and to save its state to another side more secure, where it can continue its execution. By *soft mobility*, we mean a system where code can migrate from one site to another site. Many reasons can cause such migration and many methods and techniques can be used. On the other hand, travelling users who request some computing services need also some specific mobile devices. In this last case, we talk about *hard mobility*. Solutions that study and consider soft mobility are classified as **mobile code** studies. Solutions dealing with hard mobility concern **wireless and mobile networking**. The most of current problems in computer science and also in telecommunication fields require hybrid solutions where soft and hard mobility are employed together.

Mobile Computing covers the two fields of Mobile devices and Mobile codes. Mobile computing knows an increasing use in computer science and in telecommunication field. The 90^{th} decade was known by the development and a particular concentration on *mobile code technologies* [51]. In this period, researches on mobile code reached a maturated state. This maturity allows researchers in the domain to propose many technologies as programming languages and developing environment supporting mobility aspects. These programming languages and environments spread mobile code technologies and make it one of the basic solutions in computer science design methods. The second decade (after 2000) is more known by the growing of *mobile and wireless networking technologies* [50]. With the development and the elaboration of mobile computing technologies, some unresolved problems can be resolved but also some traditional problems can have more sophisticated and adapted solutions.

In software engineering, the development of any solution is realized through a set of developing steps. Each step produces some middle-product that can be used as input for the following step. The final product will be the requested solution as a software system. These steps start always by an analysis of the requirements, the current problems, and the available technologies and solutions. This analysis opens road to a series of other phases. Briefly, a design step offers the architecture of the system and the set of its modules, then a coding step that transforms this architecture to a real system ready to be executed, finally test activity is used to find errors and debug the system. The poorness of the test activity requires two other activities: **verification** and **validation**. To ensure that the system realized is correct, **verification** activities can be included into the design steps or after the coding of the system. To insure that the system conforms to the first requirements defined by the future users, **validation** activities must also be achieved between steps of the development process and on the final product of the development. The middle-products can be described using **formal languages**. A formal language uses a well defined syntax to describe the product, and has a formal semantics. This formal semantics allows the developer to apply **formal verification**. When a formal language is used in a developing process, some of the products are presented as formal description that we call **formal specifications** of the system. In a **Complete Formal Approach**, all steps of the development are realized using formal tools. A formal language is used to present the result of the analysis, the architecture of the system is described formally,

and the architecture can be transformed formally (some time also automatically) to a code. In a formal approach, the verification can be realized through all phases (and also can be automated). **Formal methods** are methods where the developer uses some formal language to describe its product at some level as a formal specification, and then apply some technique to verify some properties in this specification. The use of formal methods was proposed with the first proposition of software engineering as a discipline to guide developers to produce high quality software. However the use of formal methods was always limited. This is due, sometimes to the difficulties that seems as a disadvantage of these methods, and sometimes to the non-conscience of developers of the power of such techniques to produce software with no errors. Though these problems, researches on formal methods were not stopped, many formal languages and many verification techniques were elaborated, developed, and used in some applications. In the most approach of software engineering, we consider that a system is always a combination of a set of **structures** (representing data used by the software) and a **dynamic** that represents the execution of the system. Following this last consideration, formal methods were proposed to specify and verify even the structure of the system or the dynamic of this system. Some formal methods are more general and allow the developer to specify the structure and the dynamic of the system. The idea of mobility in computer science imposes challenges to the developer in many levels and on many aspects. Programmers were interested to proposing technologies that allow the implementation of mobility in computing, and their efforts make it possible to implement many ideas that seem fantasy firstly. Designers were interested to propose approaches, and design/specification languages to guide developers in this new area. On the other hand, researchers on formal methods, tried to give their part in this evolution.

To specify mobility in computing, some researchers propose the use of classical formal tools and methods. They argue their choice to the maturity of these classical techniques and their richness on the verification aspect. Other researchers consider that classical formal tools and methods are poor and have not the expressive power to handle a complex aspect as mobility. These researchers consider that the specification of mobility requires new formal methods or at least an important update of the classical methods. The motivation behind the proposition of a new formal method for mobility is always to give a tool with more expressiveness or a tool that can treat some aspect not considered by existing tools. Through our investigation in this research, we have concluded that formal methods proposed in mobile computing can be divided into three families: methods based on **Processes Algebra** calculi [9], those based on **Petri Nets** [27], and finally someone which were based on **Rewriting Logic** [39]. In a *Processes-Algebra*, processes are considered as algebra-operands and a set of operators are defined to manipulate these processes as their operands. The CCS [1] (calculus of communicating systems) is the famous classical formalism proposed as a processes-algebra. This algebra has been extended to handle mobility, the obtained calculus is π-calculus [2]. With CCS, one can specify a set of processes that communicate through a set of gates. These gates are used as a synchronization tools between processes. In the π-calculus, a sender uses its output gate to send some information to a receiver; this last one receives the information through its input gate. The idea in the π-calculus is that the communicated information can be the name of a gate. So a sender can send to a receiver the name of a gate. The receiver can use this new name as an input or output gate. With this idea, the π-calculus can be used to specify a system where its configuration can be changed during its execution. The π-calculus is more adequate to specify systems where processes are fixe, but their

communication gates can be updated through the execution of the system. To give more expressiveness to this calculus, the HOπ-calculus [3] (high order π-calculus) was proposed. In this extension, processes can exchange other *processes* through gates (the exchanged processes are called agents). With HOπ-calculus, one can specify systems where processes migrate. One of the most important notions in mobility is the processes' **localities**. Mobility in a computing system can be inspected through the change of processes localities. The precedent calculi don't specify this notion. *Mobile ambient* [4], and *Join calculus* [5] are two calculi which can be considered as extension of π-calculus, and that allow to specify explicitly processes localities. In mobile ambients, agents (or processes) can be into an ambient. Agents can migrate from one ambient to another ambient; this can model a code that migrates from one device to another device. Ambients can be structured hierarchically. An ambient can surround a set of ambients, and an internal ambient can change its locality (its surrounding ambient). Migration of ambients can model hard mobility in wireless and mobile networks. The join-calculus is proposed as a calculus for mobile agents programming. In this calculus, agents are called locations. Locations can migrate and they are transparent. The transparency of locations makes interactions between them implicitly. A location can interact with another location if it knows its name. Other extensions of these calculi were proposed to deal with some specific aspects of mobility like: sπ-claculus [7], and sjoin-caclulus [10], proposed to deal with the security aspect in mobile computing. The probabilistic mobile ambients [6] is an augmented version of mobile ambients. If in mobile ambients, the behavior of the modeled system is nondeterministic; in this augmented version the behavior can be also probabilistic.

Petri Nets represent another formal approach adopted by researchers to develop specification and verification techniques for mobile computing systems. A Petri Net (PN) [27] is a bipartite graph where nodes can be *places* or *transitions*. These nodes can be connected by a set of *arcs*. This graph describes what we call the **structure** of a Petri net, which never changes. The places in a Petri net can be marked with tokens. The set of tokens distributed on these places is called the marking of the Petri net. The transitions in a Petri net can be fired if some preconditions are satisfied. When a transition is fired, the marking of the net is updated. Some tokens disappear from some places (input places of the transition) and other tokens appear in some other places (output places of the transition). Firing transitions and updating of the marking of the Petri net describe the **dynamic** of the Petri net. In their originality, Petri nets were proposed as a formalism to describe the dynamic of concurrent systems. The classical model is so simple and with a poor expressiveness. This model was extended to solve some problems inherent to the classical one and to deal with new systems that require more expressive power. Colored Petri Nets (CPN) [31] was proposed to give a formalism where tokens can be structured data. More complex systems can be modeled using CPN and where the model stays always controllable. The use of Petri nets to model mobility requires the development of new versions of this formalism. The classical one seems to be so poor to deal with the aspect of mobility. Researchers have identified early that the most problem using Petri net to model mobility is that PN model only the dynamic of the system, though in mobile systems it is no more the dynamic, only is changing, but the structure also changes over time. The most proposed extensions of PN to model mobility have as objective to give to the formalism the expressive power to model a system where its configuration changes during its execution. Through our study, we have concluded that works on extending Petri Nets to model mobility can be classified into three classes. This classification is based on the degree of dynamicity allowed to reconfigure the graph of the Petri Net. In the first class (for example

in [25]), the graph does not change. So there is no reconfiguration of the graph, and mobility is only simulated on the graph of the Petri net. In the second class, even the graph doesn't change the tokens are not simple data. Tokens can be themselves nets (for example the works [21, 22, 23, 29, 30]) or tokens can be algebraic expressions (like in [26]). Mobility of agents is modelled by the movement of these tokens, and behaviour of mobile agents will be modelled by executions in these tokens. A third class of extensions is the one where the idea is to allow a modification in the graph of the net during the firing of some transitions (as examples in [19, 20, 24]). In this kind of extension, the structure of the net can be modified and this modification will be used to model dynamicity in mobile computing systems. In the three following paragraphs, we present in more details these kinds of extensions.

Predicate/Transition nets (PrNets) [25] is an extension of CPN for mobile agents. In this formalism, a set of agents with common behaviour can be modelled as a net (*a template*). A location is modelled as *system net* (a set of template nets). Connectors are used to connect locations, and to allow the migration of mobile agents. Migration of an agent is modelled by the transfer of some token from the local location to the destination one, through a connector. This transfer of token will make the agent inactive in the first location and active in the destination one. With PrNets, authors give a technique to simulate agent's mobility through token transferring between interconnected PrNets.

In Object Petri Nets [30], and Elementary Object Nets (EON) [21], tokens can be Petri nets themselves. In an EON, we distinguish between *System Nets* and *Object Nets*. Object Nets play the role of object tokens that can appear in places of a System Net. Here, a two-level system modelling technique is introduced. The System Net which represents the external level and the Object Net which represents the internal level can have some synchronous transitions. In this case, these transitions must be fired simultaneously in the two levels. Object Nets used as tokens in a System Net can also interact. EON formalism was proposed to model some kind of systems like: workflow, flexible manufacturing, and mobile agents. Based on the EON formalism, other proposals were done: Nested Nets [22], Petri Hypernets [23], Nets within Nets [29], … etc In Nested Nets [31], firing some transitions creates new nets (called *token nets*) in their output places. Nested nets are also hierarchic nets, where we have different levels of details. Places can contain nets, and these nets can contain also nets as tokens in their places. This formalism was proposed to adaptive workflow systems. Adaptivity means an ability to modify processes in a structured way, for example by replacing a subprocess or extending it. Petri Hypernets [23] are proposed to model mobile agents. Mobile agents are modelled as nets. These mobile agents are manipulated by other agents (modelled as nets) who can be also mobile. We call *Open Net* a net used to model a mobile agent. This open net plays the role of a token in another net; this last one is called *hyper-marking Net*. As a difference with Valk's proposal [21, 30], the inter-level synchronization in Hyper-Nets is achieved solely by means of exchanging messages. PEPA nets [26], are a combination between Coloured Stochastic Petri Nets [32] and the Stochastic Processes Algebra PEPA [8]. The PEPA algebra can be used to specify systems composed of a set of components that can be concurrent and that can cooperate. These components will be called PEPA components. Stochastic variables are used to specify the durations of activities executed by these components. In PEPA Nets, the set of colours used in the definition of the net are a set of a PEPA Components. In [26], authors present an example of modelling a mobile agent system using a PEPA Net. Places of the PEPA Net are used to model the set of hosts that the agent

can visited. Transitions in the PEPA Nets model the activities that the agents can execute. The agent itself is modelled as a token specified as a component in PEPA algebra. In this formalism, the structure of the PEPA Nets does not change. The mobility is modelled through the firing of some transition, and the transfer of one token (an agent or a process) from one place to another.

Self Modifying Nets (SMN) [28] is an extension of Petri Nets. In this formalism, edges can be labelled by names of places. If the name *p* is used as the weight of an arc, then this means that the number of tokens to be moved through this arc is equal to the current marking of the place *p*. So, in self modifying nets, the weights of arcs are dynamic. These weights depend on the current marking of the net. Even SMN offers more computational power than PN; mobility was not the objective of this extension. Though, this formalism was the basis for other more important formalisms like "Reconfigurable Nets" [19]. In Reconfigurable Nets, a set of rewriting rules are used to modify the structure of the net. In this formalism, the set of transitions does not change; however, some places can appear or disappear. Appearance and disappearance of places change the interconnection of the net. This formalism is proposed to model nets with fixed components but where connectivity can be changed over time. This formalism was also proposed for dynamic workflow systems. A dynamic workflow system retains the same set of tasks, but the order in which these tasks are executed can be changed over time. Reconfigurable Nets can be translated into Self Modifying Nets. In [24], authors proposed Mobile synchronous Petri nets (MSPN) as formalism to model mobile systems. MSPN is an extension of CPN. In this formalism, two concepts are introduced: *Nets* (an entity that can model an agent or a process) and disjoint *Locations* (which can model environments). A *net* can change its location when some transition is fired. To explicit mobility, the marking of a *net* in a MSPN is extended with the information of the current location of the net. The marking of a *net* contains both: marking of all places as in ordinary CPN, and the identifier of the current location of this *net*. To change the location of a *net*, this *net* must contain a specific transition called *go*. Firing a *go* transition, in a net, moves this one from its locality towards another locality. The destination locality is given through a token in an input place of the *go* transition. Finally, Mobile Petri Nets (MPN) [20] extend coloured Petri nets to model mobility. MPN is inspired from join-calculus [5]. In this formalism, the set of output-places of a transition can be modified during the firing of this transition. A transition can change its output places. The input-expressions of a transition define the set of its output-places. In the same work [20], authors present Dynamic Petri Nets (DPN) which is an extension of MPN. In DPN, the firing of a transition can add a new subnet to the original one.

Beside the use of process algebra and Petri net, the *Rewriting Logic (RL)* [39] was also exploited to specify mobility. Firstly, *Rewriting logic* [39] was proposed as a unify framework for all formal models dedicated for concurrency. Rewriting logic uses algebraic data type to specify distributed systems. An algebraic data type is presented as an *equational specification*. This specification consists of a *signature* (types and operators) and a collection of conditional equations. The dynamic of the distributed system is then specified by set of *rewrite rules*. *Rewrite rules* model actions that transfer the system from one state to another state. Theoretical ideas developed in RL were implemented in *Maude* language [37]**.** Maude allows the verification of specifications written in rewriting logic. To handle mobility, a *Mobile Maude* [38] was proposed. *Mobile Maude* is an object oriented high level language with asynchronous message passing. *Mobile Maude* introduces two new concepts: *processes* and

mobile objects. A process models the computational environment where one or more mobile objects can reside and can be executed. Mobile objects are instances of classes that can migrate from process to process and can communicate by message passing. The key feature of mobile Maude is that it deals with various security aspects: Cryptographic authentication which protects machine from malicious mobile code, redundant checks which insure reliability and integrity of computations, and public-key infrastructure service which protects communications.

In our study, we were interested to use Petri Nets to model and verify mobile agents. We were interested to propose extension of Petri Nets where the structure of the net can change during its execution. So we can classify the set of our works in the third class of previous works. We have proposed several ideas presented in some papers. In [42], we have presented *Labelled Reconfigurable Nets* (LRN). In this formalism, we have extended Petri Nets with labelled reconfigure transitions. When a reconfigure transition is fired, it changes the structure of the net. The change that occurs depends on some information (destination of migration, type of migration, required and type of resources for processes or agents … etc). This information is presented in a label associated to the reconfigure transition. With this idea, we have showed how some basic concepts defined in mobility can be modelled like: Mobile Agents, Remote Evaluation, and Code on Demand. This formalism was extended to handle time property in [43, 44]. In [45], we have extended LRN toward *Coloured Reconfigurable Nets* (CRN). In CRN, we have used structured tokens to compute information required to mobility and so we have abandoned the idea of labels. Using structured tokens makes the model more flexible because the information can be computed and so can change over time. These previous formalisms are so naïve and they can model explicitly and easily mobility of agents, but once trying verification, we have concluded that expressiveness power of these models makes verification so hard. In [46], we have proposed a technique for modelling and simulation of mobile agents using LRN and an extension of Maude [37]. This extension is called *Reconfigurable Maude* (RM). In RM, we allow the possibility of changing the structure of Maude theories, during their execution. We have introduced specific rules called *reconfigure rules* (RR), that once executed can change the structure of a Maude theory. A RR can have its effect on the local theory where it is defined, or it can have an over-effect on other theories. RM theories can be distributed on a local net. A prototype of RM was realized with some students. The idea was to firstly model mobile systems using LRN, then translate these nets into RM, and finally simulate the execution of these specifications using the realized prototype. The basic idea behind these works was always the give to the net some mechanism allowing it to reconfigure its structure when some transition is fired. This reconfiguration is done by adding some component (places, transitions, arcs) or deleting them from the original structure. The components to be add or delete must be defined some were as labels or as tokens. The problem of these formalism stays in the verification phase. Each time we try to allow the most and fine reconfiguration possible in the net, the analysis will be harder and sometimes impossible. Recently, we have exploited the *Dynamic Petri Nets* [20], which makes it possible to add places and new nets to the original net when transitions are fired. The reconfiguration in DPN is not general and must respect several conditions. Our objective was to present a formalism more flexible than DPN and in which the structure can be changed by adding or deleting any component. We have, so, proposed the *Flexible Petri Nets* (FPN) [47, 49] as a generalization of DPN. The FPN is the more flexible formalism based on Petri nets. Reconfiguration can be done in fine granularity, and in a free way. This

quality makes the formalism the more adequate to model the most sophisticated Reconfigurable Systems (Mobile computing systems, mobile networks, mobile robots …). One of the ways to analyse FPN model can be the translation of these models into some DPN very complex. These DPN models can be themselves translated into CPN, like shown in [20]. We consider that our proposition about *Flexible Nets* seems to be the most elaborated extension for Petri Nets with reconfigurable structure. Using this formalism, the designer of reconfigurable systems has a tool with an expressiveness power, which makes formal modelling easy.

This thesis is composed of four chapters. The first two chapters present a state of the art, and the two last chapters present our contribution. We adopt the following organization:

Chapter 1: Mobile Computing. This chapter presents a state of the art on mobile computing;

Chapter 2: Formal methods for mobile computing. This chapter gives in more details some of the most important works in the domain of formal methods applied in mobile computing. We will restrict the presentation only to the works that have inspired us, and which have influenced our proposition;

Chapter 3: Extended Petri Nets: This chapter present our contribution at the *modelling level*. The chapter present a set of extensions proposed to model mobility and in more general reconfigurability in systems. For each extension, we will give the definition of the formalism, its characteristics, some examples, and the analysis issues.

Chapter 4: The Encoding of Flexible Nets. This chapter present our second contribution at the *formal analysis level*. We will present the translation of Flexible nets into Dynamic Petri Nets, and we will prove this translation.

This thesis will be concluded by a conclusion in which we present a self-evaluation of our work, present more comparisons with other works, and finally show some perspectives of the present work.

Chapter I: Mobile Computing

1 Introduction

With the apparition of distributed systems, the development of wireless computer networks and the diffusion of portable devices, we experienced the emergence of mobile computing. In this context, mobility concerns either movement of hardware devices or migration of software applications. The purpose of this chapter is to present a state of the art on mobility. One attempts to trace the way in which ideas have evolved. However, if we start with a broad aspect of mobility, we will focus more and more on a particular axis: *mobile agents*. These are autonomous software entities, performing tasks to the profiles of their owners.

In section (2), we begin by exposing definitions of some principal concepts in the field: mobile computing, mobile code (or code mobility) and mobile agents. Motivations of mobility with its both hardware and software versions will be presented in the third section. The fourth section focuses on soft mobility and explores the origins of this idea. Considering the concept of mobile code systems or mobile system components as a unifying concept of different forms of soft mobility, section (5) presents an architectural vision (adopted in several works) for such systems, as well as mechanisms derived below this architecture.

Section (6) presents some of the most relevant existing technologies (programming languages and platforms). These technologies can be used to implement solutions based on mobile code. However, such solutions must first be designed, and Section 7 presents conceptual paradigms derived from the architectural vision presented in Section 5. These conceptual paradigms can be implemented with different technological solutions, even if some solutions are more suitable for specific paradigms. Section (8) describes the potential scopes of mobility, and then in section (9) we mention some problems inherent to mobility. Despite these problems, theoretical ideas have passed to industrial and commercial realizations, which are completed or in progress. So, in section (10) we present a variety of these realizations. Before concluding this chapter, section 11 refers to international standardization efforts. The goal of these efforts is to bring near the existing heterogeneous implementations and to avoid future discrepancies.

2 Definitions

In the domain of mobility, three concepts are common: *Mobile Computing, Mobile Code* (or code mobility) and *Mobile Agents*. We present below a variety of definitions of these three concepts.

Mobile computing: *"Paradigm in which users carrying portable devices have access to a shared infrastructure independent of their physical location"*. [58]

Code Mobility: *"Code mobility can be defined as the capability to dynamically change the bindings between code fragments and the location where they are executed"*. [54]

"Mobile code are soft-ware that travels on a heterogeneous network, crossing administrative domains, and is automatically executed upon arrival at the destination …". [80]

Mobile agents: *"Mobile agents are programs that can move through a network under their own control, migrating from host to host and interacting with other agents and resources on each".* [60]

"A mobile agent is not bound to the system where it begins execution. It has the unique ability to transport itself from one system in a network to another. The ability to travel, allows a mobile agent to move to a system that contains an object with which the agent wants to interact, and then to take advantage of being in the same host or network as the object". [59]

"Computations that are able to relocate themselves from one host to another". [73]

"A piece of code and its associated data moving about executing autonomously on behalf of its owner". [63]

"Mobile agents are software abstractions that can migrate across the network representing users in various tasks… A mobile agent has the unique ability to transport itself from one system in a network to another in the same network." [53].

We consider that the first concept "*mobile computing*" concerns devices mobility (we call it hard mobility). The other two express the mobility of applications (software mobility). The novelty in the case of agents is that mobility is due to internal determination (autonomy of decision). Agents move, without external intervention.

3 Why Mobility?

The two variants of mobility (hard and soft) have important motivations:

3.1 Motivations for Hard Mobility

It concerns the movement of computing devices (laptops, PDA: Personal digital assistant, mobile phone ...). It is motivated by:

• The diffusion of wireless networks;

• The diffusion of cellular telecommunication networks. The devices used in these networks (mostly cell phones) acquired day by day computing capacity and processing important information;

• Mobile users: requiring computing mobile devices.

3.2 Motivations for Soft Mobility

Movement or migration of a code (a process, an object or procedure) may have different reasons. According to [64], such a migration can provide:

• A load balancing between processors;

• A performance in the communication: by bringing together objects that communicate intensively on the same nodes;

• An availability of the required objects in the nodes where these objects are requested;

• Make services available on a node: by downloading these services;

In addition to these reasons, the authors in [54] and [51] suggest other motivations:

• Need for scalability in large networks;

• Need for specialized services for a wide audience (networks WANs): Number of customers grows rapidly and their needs differ. Predefined servers with a minimum of specialized services may provide specific services through their enrichment by downloading components;

• The dynamic nature of telecommunications infrastructure: These infrastructures are characterized by a low flow and low reliability (frequent disconnects);

• Deployment and maintenance of system components on networks: In large scale networks, components (with expertise on the maintainability and reconfiguration) run on the network and visit the various nodes to do the needful;

• Creation of stand-alone applications: such applications run on the net, to provide complex local interactions. These interactions are not possible with low-speed infrastructure and frequent disconnections;

• Flexible Data Management: Data (accompanied by codes necessary for their treatment) convey the network. Nodes in a network receive the data and their treatment protocols (protocols encapsulation);

• Improve fault tolerance in distributed applications: Move existing applications from their site, if it is subject to failure;

3.3 Motivations for Mobile Agents

Considering that an agent is an autonomous code, some characteristics of these agents promote their use than other paradigms. In [63], we find the characteristics of mobile agents that promote their use in mobile networks:

• They can migrate from one mobile computing device to a network for collecting information and utilize resources. It is more effective to put itself in the network (resources and information) than to send requests and wait for answers. During its work on the network, the agent is disconnected from its source node. It may return with the results, later, when its node will be connected;

• No obligation to handle network faults (except when migrating);

• They do not require pre-installation of applications on specific hosts;

• Surpass the rigid model of client-server to a peer-to-peer model. This last one seems to be more suitable for program for which needs change (moving from a client to a server and vice versa);

• Scalability of applications: Move the job to the most appropriate node;

• Experience shows that mobile agent's paradigm is easier to understand than others;

The authors in [66] offer seven good reasons for using mobile agents:

• They reduce the network load: Move the code to the service, once, then all interactions will be local. Move the code to the data, if this data is huge to move.

• They overcome the latency of a network: In real-time applications, latency becomes significant and unacceptable with large networks. Mobile agents can migrate to the components (i.e. robots) and insure a local control;

• They encapsulate protocols: Mobile agents can move the appropriate protocols to the sites concerned (i.e. to receive or transmit specific data formats). They enable efficient updating of protocols;

• They run independently and asynchronously: For mobile devices with frequent disconnections, mobile agents may migrate out of these devices towards the network. These agents run on the network asynchronously and autonomous. Once achieving their tasks, these agents return to their devices during a subsequent connection;

• They adapt dynamically: They feel the properties of their environment (the non trust network segments, the segment where the flow is low ...), then they reconfigure themselves to get an optimal configuration (load balancing ...);

• They are heterogeneous in nature: They are independent of the hosts and transportation. They depend only on their execution software environment (of specific platforms such as abstract or virtual machines, like the JVM). They offer a good model for heterogeneous systems (hardware or software);

• They are robust and fault-tolerant: if a host risks falling down, mobile agents can migrate from this host.

4 Origins of the Soft Mobility Idea

The idea of moving code from one machine to another machine is not new. It dates from the seventieth. In the literature, one can find several innovative ideas: "*Remote batch job submission*" [86], the *rsh* command of UNIX, and the printer language *PostScript* [101]. For example, to print on a PostScript printer, the printer requires a PostScript program describing all the properties of the printed page. This program will be sent to the printer, where it will run. Other examples are: the processes migration in the *Sprite* System [55], The *Apiary network* for knowledge base systems [61], and transparent migration of active objects in distributed systems (i.e. *Accent* kernel [76], the *Eden* system [71], *DEMOS/MP* [75], *Emerlad* [64], *Chorus* [77], *Locus* [79], *V-system* [78], and *Cool* [69]).

According to [52], novelty in current mobile systems compared to these backgrounds can be found in three ideas:

• We are interested in large systems like the Internet (WAN), rather than the traditional distributed systems LANs;

• Mobility is under the control of the programmer (Programmer is AWARE). Mobility is not transparent;

• The goal of mobility is not only the load balancing.

Systems with these qualities are called *Mobile Code Systems* (MCSs).

5 Architectures and Mechanisms for MCSs

Running a MCS requires the presence of hardware/software architecture. Considering the differences between the modern MCSs and classical mobility systems, [51] proposes a new architecture of a soft/hard environment to implement MCSs.

5.1 Architectures for mobility

In [51], authors have presented two types of logical architectures that can be used to implement code mobility (**Figure I.1**).

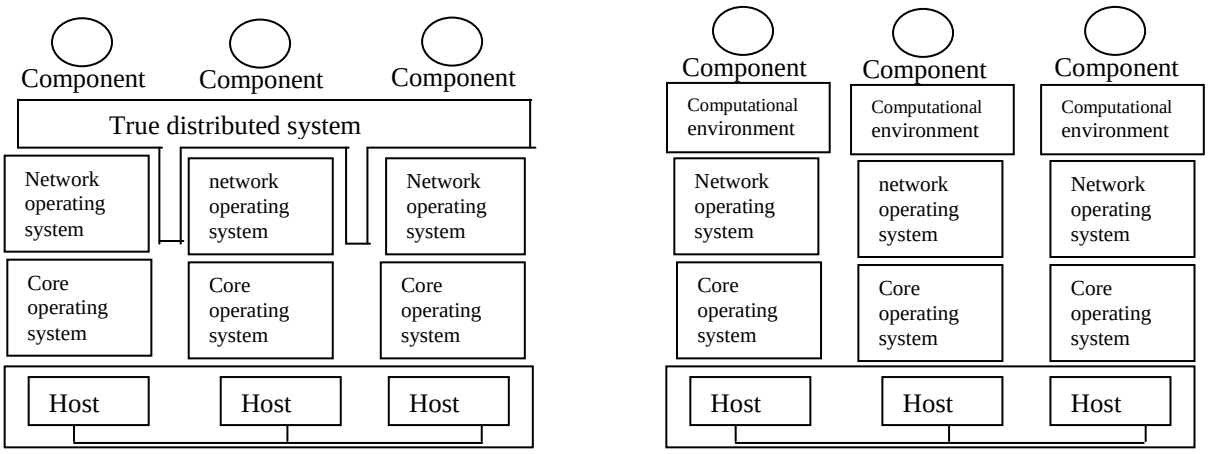


Figure I.1. Architectures for code mobility systems.

In both alternatives, the proposed architecture is composed of five layers:

• Layer 1: “hardware” consists of the hosts (nodes) and communication network infrastructure;

• Layer 2: “Core operating system”, the core operating system for each host. This layer should provide the basic functions: files, memory and processes management;

• Layer 3: “operating network system”, this layer provides the functions of network management. In this layer, the communications are not transparent. The sockets service is an example of services to be provided by this layer;

• Layer 4: it must provide the mobility service. It can be either a "*true distributed system*", which provides the functions of a distributed system with transparent communication and mobility or a "*computational environment*" where communication and mobility are not transparent;

• Layer 5: “component”, this layer includes both resources (logical or physical) and calculations (programs, processes or applications) called "*executing units*” (EUs).

Each E.U is composed of:

• The *code segment*: the source code. Static part of the unit;

• The *state*: composed of a *data space* (references to resources available for the EU. These resources can be local or remote) and an *execution state* (consisting of private data, the stack of calls, an instructions pointer ...)

A MCS must be executed on an architecture of type 2 (with a computational environment, so, mobility is not transparent with respect to a programming vision).

5.2 Mechanisms for mobility

Considering the preceding vision, code mobility means moving an EU (in entire or some of it parts). In [51], we find an exhaustive classification of possible mechanisms for mobility, and how to how to manage bindings (or links) between the EU and its resources after migration from its source computational environment (CE).

5.2.1 Kinds of mobility

Based on this criterion, we find two kinds of mobility: strong mobility and weak mobility.

1. **Strong mobility**: consists on moving the code segment and the execution state. In this class, we have two sub-mechanisms, *Migration* and *Remote Cloning*:

Migration: firstly, the EU is suspended on its local site, and then it is transferred to the destination CE where it will be restarted. This migration may be *pro-active* or *reactive*. In *proactive migration*, the destination and timing of migration are defined by the EU. In *reactive migration*, time and destination of migration are not defined by an EU, but they are defined external (i.e. EU manager).

Remote Cloning: consists to create a copy of the EU in the CE, where the EU wants to move. This mechanism can also be *pro-active* or *reactive*.

2. **Weak mobility**: in this kind of mobility, only the code segment can be moved, perhaps with some initialization data needed to restart it. In this class, the mechanisms can be classified according to four criteria: (i) Direction of the transfer, (ii) Nature of the transferred code, (iii) Synchronization between the source CE and the destination CE and (iv) The time of restarting of the code after its transfer.

2.1. ***Shipping***: the CE source sends the source code to the CE destination. The transfer can be *standalone* or *code-fragment*.

Stand-alone code (run on the fly): in this kind of transfer, the code creates its own EU, which will execute it. In this case, the source CE and the destination CE can be *synchronous* or *asynchronous*. In the first case, the source CE is suspended waiting the termination of the execution of the transferred code on the destination CE. In *asynchronous* case, the source CE is not suspended. In this case, the transferred code may start running on the destination CE *immediately* or *differently* (waits an event or condition, to get started).

Code-fragment: in this case, the execution of the transferred code is realized in the context of an EU that exists on the destination CE.

2.2. ***Fetching***: a CE (Destination) downloads a code from its source CE. Like in fetching, we can find the same kinds of transfer.

5.2.2 Managing bindings

Before its movement, the EU uses some resources. These resources can be local (on the local node) or remote (on a remote node). These resources can be *private* to the EU or *shared* with other EUs. These resources may be *transferable* or not *transferable*. The EU should have links (references) to such resources. These references can be of three levels:

- *Reference by identity*: it is the strongest link. The EU must always have access to that resource to run.
- *Reference by value*: EU is interested at the content of the resource not to the resource itself. If it is possible to have a copy of the contents of this resource, the EU will be satisfied.
- *Reference by type*: what is important is the type of the resource. A resource with the same type can satisfy the EU.

Let DCE be a destination CE, U an execution unit, R a resource, and B the type of reference between U and R. According to these different levels of references and the type of resource (transferable or not), there are different mechanisms for managing links:

- *By move*: transferring R with U to DCE, without changing B.
- *By Copy*: creating a copy of R on DCE and update B.
- *Network reference*: R is not transferred and B will be updated to ensure the link between U and R.
- *Rebinding*: a resource R' of the same type that R exists on the DCE. After that U passes to the DCE, B is updated to ensure the link between U and R'.

Depending on the type of the resource and the type of binding between the EU and the resource, The Table I.1 shows how these different mechanisms can be used.

Type of resource / Type of reference	transferable	Non transferrable
By identity	*By move* and *Network reference*	*Network reference*
By value	*By copy (by move* by *network reference)*	*Network reference*
By type	*Rebinding* (or one of the other mechanisms)	*Rebinding* (*network reference*)

Table I.1 Mechanisms for managing links

6 Programming Languages

Currently, there are several programming languages (more than 100) and a variety of platforms for the development of MCSs. Some of these systems consider these codes as mobile agents. The purpose of this section is not to present all these platforms, but we want to discuss some of the most important ones. We will present some examples that provide strong mobility (like *telescript*) or weak-mobility (like *Aglet*), and which are object oriented (*Java*) or functional (*Objective Caml*).

6.1.1 The Emerlad system

Emerlad [64] is one of the first languages supporting migration of objects. *Emerlad* is an object system (supporting the concept of abstract data type). Travelling objects can be simple objects (data) as they may be processes. *Emerlad* runs on a LAN of 100 nodes (maximum) homogeneous and trusts. It is even possible to move the arguments for a remote call. Mobility is expressed with primitives available to programmers (i.e. **move object to node**).

Objects (with their operations) are distributed on the nodes of the network. *Active objects* (accompanied by threads) can make calls to operations of other objects on remote nodes. The execution of these operations creates a stack (below this stack, the process, over this process, we can have records of this process where the operations of other objects are invoked).

A remote call is as follows:

- Either the calling process migrates completely to the node of the invoked object.
- Or just the part appealing the process migrates to the node of the object being invoked.

On the latter, a new stack is created, and thus a new process (A *fine granularity*).

6.1.2 Telescript

Telescript [83, 84] developed by General Magic Company is the first commercialized languages. It is an object-oriented language which provides *strong mobility*. *Telescript* is not a general programming language, but it is dedicated to specific applications (communication). The central concept in *Telescript* is the *agent*. An agent moves autonomously in a *remote-sphere* to conduct business profile client. A *remote-sphere* is a set of execution engines (*Telescript* interpreter and a *place*). The *places* are stations where agents can be received. The user can create such places.

A *Telescript* agent is a process characterized by:

1. The *telename* composed of two components: authority and identity.
2. The *owner*, the possessor of the new objects created.
3. The *sponsor*, the process that the authority will be attached to new created objects.
4. The *client*, the object whose code calls the current operation.
5. *Permits*, specifies the capabilities of this process.
6. *Age*: maximum lifetime in seconds of the process.
7. *Extent*: maximum size of the memory area where the process will be executed.
8. *Priority*: used to decide when to run the process.
9. *canGo*, *canCreate*, *canGrant*, and *canDeney*: boolean determining whether this process can move, create, increase or reduce *permits* of an another process.

Telescript provides a primitive **go** that agent executes to move to another host. Security is provided by the concepts of *capabilities* (permission) and *authentications*. The capabilities define the rights of an agent; the authentication allows the sites to accept only some agents (those authenticated). The agent is running on a virtual machine that detects any violation or illegal instruction. The exception mechanism allows agents to cover certain errors.

Telecsript is the first commercialized system. It has been used for network management [60], active e-mail, e-commerce ... It supports host mobility (hard mobility). It was used on PDAs (Personal Digital Assistant) as the case of Sony Magic Link.

6.1.3 Agent Tcl

Agent-Tcl [85] is a language that provides a transparent migration of agents (the agent state is collected by the system, transparent to the programmer). An agent can migrate between two mobile machines, or between a mobile machine and another fixed machine. Agents (called *scripts*) can be programmed in several languages (*Agent-Tcl*: an extension of Tcl "Tool Command Language" (http://www.ensta.fr/~diam/tcl/) or ***java***).

Agent-Tcl is a system composed of a set of servers on the hosts of a network. One statement: "**agent_jump**" captures the agent state, encrypts, signs it and sends it to the server on the destination host. The server authenticates the agent and starts a ***Tcl*** interpreter to execute this agent. The ***Tcl*** interpreter restores the state of the agent and restarts the execution of the agent from the statement that succeeds "**agent_jump**". *Agent-Tcl* is not object oriented, so actually it is a migration of procedures.

The statement "**agent_meet**" sent form an agent and accepted by another, allowing both to communicate with low-level primitives. A high level communication is possible through the mechanism of ***ARPC*** (Agent remote procedure call [74]). This language provides security in an uncertain world (as in internet). It was originally used for information retrieval, then for work-flow.

6.1.4 Java

Java [52] developed by Sun is a class-based language, object oriented. A general language, whose objectives were portability and security. *JavaSoft* (http://www.javasoft.ch/), which is a collection of Java libraries for different purposes, has created the model of the applet (mobile code) downloaded and executed automatically when visiting a web page.

6.1.5 Objective Caml

Objective Caml [70] is developed in the INRIA (Institut National de Recherche en Informatique et en Automatique: http://www.inria.fr/). It is a functional language like based on ***ML*** (http://www.lfcs.inf.ed.ac.uk/software/ML/), enriched with object-oriented paradigm. This language was used for the development of the web browser MMM (http://pauillac.inria.fr/mmm/) in the INRIA. MMM allows linking and execution of applets dynamically.

6.1.6 Aglet

Aglet [68] is developed by IBM Tokyo Research Lab. An ***aglet*** (***agent-applet***) is an enhancement of ***java applets***. *Aglet* offers a weak mobility. Agents are java threads. It uses the concept of remote reference (***Aglet Proxy***), message passing (multicast, broadcast) and ATP Protocol (Agent Transfer Protocol) [67]. It also provides some sense of security by limiting resources for ***aglets*** (using protocols: ***SSL*** "**S**ecure Sockets Layer protocol", and ***X.501*** "http://rfc-ref.org/RFC-TEXTS/2116/kw-x.501.html").

Two primitives are proposed: *dispatch*, which sends an agent (stand-alone shipping) to a destination passed in parameter, and *retract* which, downloads an aglet (fetching stand-alone). Aglet complies with the standards MASIF [72] and FIPA (http://www.fipa.org/).

7 Conceptual Paradigms

In a development process, the design phase tries to identify the architecture (components and their interactions) of the system. Paradigms used in the design phases are independent of the technology used to implement the system. With respect to [51], designing a system with mobile components differs from the classical systems design (systems with static component). In the development of a mobile system, the sites of mobile components must be clarified during the design phase of the system. Authors in [51] have identified three major conceptual paradigms for the design of mobile computing systems: *Remote evaluation* (RE), *Code on Demand* (CoD) and *Mobile Agent* (MA).

To show the difference between these three paradigms, three concepts are introduced: *component*, *site* and *interaction*. The component is the set of *know-how*, *resources* and *computational component* (the interpreter responsible for the execution of the know-how). A site is a place where a component resides and can run. An interaction can occur between two components on the same site or two remote sites.

Let *A* and *B* be two components initially residing on the two sites *SA* and *SB*. The component *A* needs a service. *A* may have the know-how and resources necessary for this service, or some this know-how and resources can be held by *B*. The three paradigms differ according to the positions of *A* (here *A* is intuitively computational component), the know-how and the resources, both before and after completion of service. Table I.3 summarizes these differences:

	Before the completion of the service		After the completion of the service	
paradigms	S_A	S_B	S_A	S_B
Client-server	*A*	Know-how, ressources, *B*	*A*	Know-how, ressources, B
Remote evaluation	A+know-how	Rssources, *B*	*A*	Know-how, ressources, *B*
Code on demand	*A*+Ressource	Know-how, *B*	A+know-how+ressources	*B*
Mobile agent	*A*+know-how	ressources		*A*+know-how+ressources

Table I.2. Conceptual paradigms for mobility

The client-server paradigm is not a paradigm for mobility. Its presence on the table is for comparison only.

In addition to these paradigms, another paradigm has been proposed: the *Push Pardigm* [57], [62]. Its principle is opposed to those of RE and CoD. The idea is that a client sends a profile from the beginning to the server. Then, the server determines what treatments and when must they be forwarded to the client. At the right time, the server transmits the active treatment to the client. The advantage of this method is that the client will be relieved of this download.

8 Applications of Mobility

Different applications have been considered for systems with mobile components (MCSs). These applications use different paradigms in a chaotic and ambiguous manner. If it is clear that java applets is not an application of mobile agent paradigm (no autonomy, one direction of mobility), for some applications things are not quite clear.

Several authors report various possible applications:

Authors in [51] cite:

1. Distributed research of information [70], [59],

2. Active Documents: Active e-mail, web page (hypertext),

3. Advanced telecommunications services: video-conferencing [81], mobile users (with the potential disconnections),

4. Monitoring and remote configuration of devices: industrial processes, network management,

5. Management and cooperation in the work-flow: the work-flow defines activities, sites, relationships, time for their implementation, to achieve an industrial product. Mobile agents are responsible for conveying information between co-workers in a work-flow,

6. Active networks: flexible and dynamic networks according to application needs. Two approaches are proposed: (i) Programmable switches: dynamically extends the network. This approach is based on CoD paradigm; and (ii) Capsule approach: attach codes to the transferred packets. The node that receives the packet performs the associated code to process the data in the packet.

7. E-commerce: an agent looks in a market for catalogs, and then it returns to the laptop of a customer with the best rates available.

8. Applications deployment and maintenance of components in distributed environments [73].

In addition, [66] proposes as applications:

• Personal assistant: a mobile agent moves in the network to dispatch a request meeting the profile of its client. During this period, the client machine can be disconnected. Once it connects, the agent sends the response, the date, invited ... to the customer.

• Secure Mediation: When non-trust inter-agent negotiation, these agents can migrate and meet on a secure host, approved by all. This host should not favour any of the agents.

• Monitoring and reporting: agents are dispatched to sense events, until the establishment of information...

• Dissemination of information: the mobile agents seek and import the latest updates and install software for some client.

• Parallel processing: the agents dispatch several computing units to parallelize certain tasks.

9 Mobility Problems

On both axes hard and soft, mobility suffers from several limitations and receives a variety of criticisms. Below we expose some weaknesses that know the discipline.

9.1 Shortcomings of hard mobility

With respect to [56], the development of mobile systems meets several challenges:

Wireless communications suffer of disconnection, low bandwidth (1Mbps infrared, radio 2Mbps, 9-14 cell, by cons for 10Mbps Ethernet, 100 Mbps FDDI and 155 Mbps ATM, non-portable wireless MOTOROLA 5.7 Mbps) ...

Mobile users are suffering from problems related to variables working conditions (areas not covered, variable areas size), problems of heterogeneous networks, security issue (the wireless networks are more susceptible to intrusions).

The network addresses, configurations (parameters: available printers, server address, hourly) changes dynamically depending on location,

Problem of devices portability: low power, risk of data loss (physical damage), small interface, limited storage, ...

9.2 Shortcomings of mobile agents

According to [65] and [63], mobility remains an immature approach. It suffers from several shortcomings:

Security Issue: The hosts in a network must be protected (i.e authenticated) against the code (agents) and malicious visitors (for viruses, hackers ...). Agents must also be protected (to be encrypted) for unknown sites.

Standardization problem: Lack of a standard mobile agent paradigm. Lack of a standard infrastructure.

Coordination between agents: the arrangement of their actions in time and space. Mobility makes the existence of spatial and temporal agent two aspects not predictable.

Many technologies (in general programming languages) have been proposed (over 100). But these languages that emerge every day do not contribute to the crucial problems: security, fault tolerance, ...

Limited performance compared to traditional solutions. The time needed for the interpretation of agents and their migration.

In addition to these technical problems, the authors in [65] suggest that mobile agents suffer from non-technical issues:

No killer application: everything is achievable by mobile agents can be achieved by traditional techniques. The good point of mobile agents is that they must submit a complete and effective solution, while other techniques dealing with partial aspects of the problem,

Lack of a path to transit from current technology (client server, applets, servlets) to mobile agent systems. This transit path should be incremental and justified by a client motivation,

Problems of financial revenue of advertising sites: These sites will lose the advantage, if mobile agents can browse the net directly discovering the services offered by suppliers.

In [82], authors report ten reasons for failure of mobile agents:

Expensive and not efficient compared to conventional solutions (exp. REV),

Their design is difficult: how to identify the interactive components and how to model these interactions,

Difficult to implement: Current technologies in prototyping phase. Environments where agents are executed are unpredictable,

Difficult to test and debug: because of the distribution and mobility,

Difficult to authenticate and control: it is not clear that identity must be authenticated and how the access control mechanism must consider this information,

May be victims of attacks by malicious sites: A host can change the code of an agent. Such a change can make a malicious agent,

MA cannot keep secrets: the lack of any practical application of cryptographic mechanisms proposed,

Lack for an ubiquitous infrastructure: Existing infrastructure has proven to be vulnerable to attacks,

Lack for a common ontology: interactions and data exchanged must meet specific formats. Despite the proposals, none of them has been widely admitted,

Similar to worms: their propagation mechanism is similar to that of Internet-worms. Infrastructure dedicated to mobile agents will be victims of attacks by worms.

10 Industrial Realizations

Despite the shortcomings and doubts surrounding mobility (and especially mobile agents), [53] was able to identify fourteen projects from academia and industry in mobility domain. The most of these projects use the mobile agent paradigm. Some of these projects are completed successfully, others are still open.

1. **ActComm** (http://actcomm.dartmouth.edu/): 1997-2002 (USA) [87]. It uses the platform *D'Agents* (http://agent.cs.dartmouth.edu/). It is directed by AFOoSR (Air Force Office of Scientific Research). This project emphasizes on wireless networks and modern applications of control.

2. **AMASE**: An industrial project opened since 1998 (Deutsche Telekom, Almagne) [87]. It uses a private platform. Its goal is to make platforms supporting mobile agents used in wireless communication environments (i.e. access to multi-media information).

3. **CoABS**: Opened since 1998 (USA) [89]. It uses ***Java-Jini*** and directed by DARPA-AFRL (Air Force Research Laboratory). Its objective is to develop different strategies to achieve the maximum gain of multi-agent systems. It proposes and evaluates strategies for control which must allow to military commanders to automate large orders, decision making, control functions (search, filtering information, mission planning ...)

4. **CogVis** (Almagne): "Cognitive Vision", opened in 2001 [80]. It uses the platform ***Mobile Agent Software***. This is an industrial project (Information Society Technology). The objective of this project is to develop mobile agents with vision capabilities.

5. **DIAMOnDS** (Pakistan): "Distributed Agents for Mobile and Dynamic Services", a university project (University of *Islamabad*), 2002-2003 [91]. It uses the platform ***Jini 1.1*** In this system, agents are mobile. These agents perform tasks (communications, data-mining) for users.

6. **DILEMMA** (IST European Commission): "Digital Design and Life-Cycle Management for Distributed Information Supply Services in Innovation Exploitation and Technology Transfer", an industrial project, 2000-2002 [92]. It uses the platform ***LANA***. Agents are mediators. The objective of this project is technology transfer and innovation between European companies, research organizations, experts, public ...

7. **HAWK** (Almagne): "Harvesting The Widely Distributed Knowledge", a university project (*Stuttgart*), 1998-2000 [93]. It uses ***JAVA***. The objective of this project is the optimized information search on the Internet.

8. **LEAP** (France): "Lightweight Extensible Agent Platform", an industrial project (*Motorola*), 2000-2002 [94]. It uses the ***JADE*** platform. The goal is to provide a standard platform. This platform can run on mobile devices (PDAs), and it is expandable to accommodate other features.

9. **MadKit** (France): a project of the University of *Montpellier* (a free ware), open since 2002 [95]. It is based on the organizational model (Agent / Group / Role). Agents can be programmed by ***Java, Shem*** and ***Jess***. This platform allows heterogeneity in the architecture of agents and in communication languages.

10. **MANTRIP** (IST European Commission): "Management Testing and Reconfiguration of IP based networks using mobile software agents," an industrial project (*Solinet Germany*), 2000-2002 [96]. It uses the platform (***MAT***). The objective of this project is the design, the development, the test, and the validation of applications based on Mobile Agent Technology (MAT) for managing IP networks.

11. **MAP** (Almagne): "Mobile Adaptive Procedure", an industrial project (*Siemens*), 1989-2002 [97]. It uses the platform ***Semo***. The objective of this system is the e-help. Domestic agents interact with citizens. Software experts agents listen to such interactions, understand interactions, and try and prepare the relevant information that can assist in domestic intrusion.

12. **Mojave** (USA): "Mobile Agent Jini Environment", an industrial project (*Motorola*) 2001-2004 [98]. It uses the ***Jini*** platform. The objective of this project is to develop malleable services. They must persist in dynamic environments. Examples of applications include: network management and intrusion tolerant systems.

13. **SysteMATech** (Almagne): "System Management based on Mobile Agent Technology", an industrial project (*IVS (*http://www.ivs.tu-berlin.de/*)*, and *DFS (*http://www.dfs.de*) accisGMBH (*http://www.accis.de*)*), opened since 1999 [99]. It uses the platform *Mobile Agent Software*. The objective is the distributed management of networks.

14. **TeleCARE** (IST European Commission): an industrial project (*UNINOVA* Portugal, *Synchronix Skill*, *Camara* Spain and *Roundrose Associated* from UK), 2001-2004 [100]. It uses a private platform. The objective of this project is to develop technology solutions for remote monitoring, remote support, and thus improve the living conditions for the elderly and their families. This platform should provide a virtual community of Help for olds.

11 Standardization Efforts

A variety of mobile agent systems (platforms and programming languages) have been proposed. They are implemented using different technologies and offer specific languages for programming agents. These factors contribute to their incompatibility and limit interoperability. Standardization efforts are in place. The two most important standards are FIPA and MASIF.

FIPA: Foundation for Intelligent and Physical Agents, established in 1996. Its goal is to standardize the interfaces between heterogeneous agents. FIPA focuses on intelligence and cooperation aspects.

MASIF: proposed by a set of companies: Crystalizer, General Magic, GMD Fokus, IBM, and Open Group. MASIF (Mobile Agent System Interoperability Facility) was accepted as an OMG standard in 1998 [72]. MASIF standard focuses on the interfaces between systems written in the same programming language, but provided by different suppliers. MASIF does not address the standardization of low-level operations: Interpretation of the agents, their serialization or implementation.

MASIF addresses the following problems:

• Managing Agents: Make standard operations: Creating agents, suspension, restoration, their termination.

• Migration of agent: provide a common infrastructure to ensure the transfer agent between systems of different types of agents.

• Management of names of agents and agent systems: To facilitate the identification of agents and agent systems by applications, MASIF standardizes the syntax and semantics of names of agents and agent systems and resources localities.

12 Conclusion

With its various forms, mobility is a fulcrum of several innovations. Since the first attempts to move processes for load balancing in distributed systems, the idea has evolved. During decades of research on this topic, paradigms have been proposed and many technologies have been implemented.

The late '90s were characterized by interesting work for the development of conceptual paradigms. REV (Remote Evaluation) paradigm, COD (Cod On Demand) and MA (Mobile Agent) represent one of the most admitted classification within the community.

On the technological stage, several programming languages and platforms for development and implementation have been made. These technologies have enabled many companies and universities to set up experimental or commercial systems based on mobile components.

Despite the elegance of the idea and the popularity of this area, many think that the approach suffers from several shortcomings. Some ones suggest that this was only a failure. The defendants of that opinion argue by the lack of performance, incompatibility and the crucial issue of security. To cover these shortcomings, efforts should be undertaken on standardization and on security policies.

Some of the shortcomings presented in this chapter like: Security problems (agents attacked by malicious sites, sites attacked by malicious agent), test and debug difficulties and costs, design and implementation difficulties, fault tolerance problems ... have been addressed as problems in *software engineering* for mobile systems. Researchers in the software engineering field tried to present solutions for these problems as development methodologies and approaches. These approaches and methodologies define the process to be applied and provide tools to guarantee the design and implementation of high quality mobile systems. *Formal approaches* are software engineering approaches with a mathematical background. These approaches provide description languages to specify the system, and then give tools and methods to prove the required properties. The next chapter will present some proposed formal methods dedicated to mobile systems. Those presented in the next chapter are only the most important ones that will help us to present our own approach in the third chapter.

Chapter II: Formal Methods for Mobile Computing

1 Introduction

Applications using code mobility are increasing. Code mobility touches critical domains (military, spacial, medicine …). Such domains require that used applications insure a set of properties. Safety, liveness, no deadlock, fault tolerance, and security are example of such required properties. Using formal methods, one can develop systems and proof (or verify) presence or absence for specific properties. Formal methods are languages, tools, and approaches allowing specification and verification of systems. Formal languages are based on a well-defined syntax and a formal semantic. Their formal semantics allow developers to verify specification written in such languages. For some languages, automatic tools are proposed to edit and verify specifications. Using formal methods in code mobility is not recent. Most currently methods can be considered as derived from process algebra area [9] or state-transition systems.

The process algebra area is the field where processes are considered in a high level as algebraic structures that respect a set of axioms. One of the most and former languages for modeling mobility was the π-calculus [3]. π-calculus is an extension for CCS (calculus for communicating systems) [1]. In CCS, a system is a set of processes. These processes are concurrent and can communicate through gates. In pure CCS this communication is a simple synchronization. In passes value CCS, processes can exchange values through their gates. In an abstract vision, a system is considered as a set of indeterminist automatons. Operational semantic of CCS is given through a labelled transition system. In CCS, processes are not mobile. The idea of π-calculus is to allow processes to exchange gates. Gates used to communicate by a process can be passed as values and exchanged sent (or received) to (from) other process. The receiver to communicate can then use the received gate. The structure of the system is modified through communication of gates between processes. In monadic π-calculus, values passed must be scalar value, but with polyadic π-calculus [11], tuples of values can be exchanged through gates. The spi-calculus [7] is an extension proposed to deal with security aspect. In another extension HOπ-calculus (high order π-calculus) [52], beside data or gates, values exchanged can be also processes (agents). So with HOπ-calculus, one can specify mobility of agents explicitly. UPPAAL tool [12] can be used to edit and verify HOπ-calculus specification (reachability, safety, bounded liveness properties). UPPAAL can also be used to simulate the system. HOπ-calculus suffers of two limits. Firstly, the analyzing of HOπ-calculus specification requires its transformation in π-calculus. Secondly, in HOπ-calculus there is no location concept. Location is an inherent concept in code mobility design paradigms. To cover the second limits different extensions are proposed. Join-calculus [5] and mobile ambient [4] are extension of π-calculus where process locations are explicit in the specification.

Petri Nets [27] was proposed to model concurrent and parallel systems. This formalism has a graphic representation and a formal background. Using ***places***, ***transitions*** and connecting ***arcs***, this formalism can specify ***states***, ***actions*** and **transitions** between states through which a system evolves. With Petri nets, one can analyze behavioural or structural properties of a system. To model mobility with Petri nets, the most important contribution can be found in high level PNets. Many extensions have been proposed to adapt Petri net to mobile systems: Mobile Nets and Dynamic Petri nets [20], Nested Petri Nets [22], HyperPetriNets [23], Mobile Synchronous Petri Net [24]...

Beside process algebra methods and Petri nets formalisms, rewriting logic [39] is proposed as a unify framework for all formal models dedicated for concurrency. In rewriting logic, the state space of a distributed system is formally specified by an algebraic data type. This algebraic data type is specified through an equational specification. An equational specification consists of a signature (types and operators) and a collection of conditional equations. The dynamic of the distributed system is then specified by rewriting rules. Rewriting rules model transitions from one state to another state, during the execution of the distributed system. Maude [37] is a language proposed for writing and verifying specification written in rewriting logic. Mobile Maude [38] is an extension of Maude for mobile systems specification. Mobile Maude is an object oriented high level language with asynchronous message passing. Mobile Maude introduces two new concepts: process and mobile objects. A process models the computational environment where one or more mobile objects can reside and can be executed. Mobile objects are instances of classes that can migrate from process to process and can communicate by message passing. The key feature of mobile Maude is that it deals with various security aspects. Cryptographic authentication protects machine from malicious mobile code, redundant checks insure reliability and integrity of computations, and public-key infrastructure service protects communications. Other works like Pigeon [40] based on reflective rewriting logic [41] tries to give high-level specification language with customizability feature. This last feature allows that Pigeon specifications can be realized on various platforms. Also in Pigeon, it is possible to specify strong mobility as weak mobility by simple modification in the specification.

The objective of this chapter is not to present in detail all these formalisms, but we will be interested only by presenting those that we will need in the presentation and the analysis of our proposition which will be presented in the next chapter. For this reason, this chapter will present in more detail two formalisms, the ***Distributed Join Calculus*** [13] and the ***Dynamic Pteri Nets*** [20]. The distributed join calculus is the formalism that has inspired the proposition of the dynamic Petri nets and also will be the inspiration of our formalism. The understanding of the dynamic Petri nets and the understanding of our proposed formalism require that the reader must have some knowledge about this calculus. The dynamic Petri nets are the formalism that can be used to unfold our formalism. So one issue that we will propose to analyze our formalism is to unfold it into the dynamic Petri nets.

2 The Distributed Join Calculus

The ***Core Join Calculus*** [14] is a language that models distributed and mobile programming. It is characterized by an explicit notion of **locality**, a strict adherence to **local synchronization**, and a direct embedding of the ML programming language [15]. The join calculus is used as the basis for several distributed languages and implementations, such as JoCaml [16] and functional nets [17]. Local synchronization means that messages always travel to a set of destinations, and can interact only after they reach that destinations. In the core join calculus, the distribution of resources has been kept **implicit** so far. The ***Distributed Join Calculus*** (DJC) [13] is a bit more complex than the core calculus. The DJC allows us to express mobile agents that can move between physical sites. Agents are not only programs but core images of running processes with their communication capabilities and their internal state.

In the DJC, a **location** resides on a physical site, and contains a group of processes and definitions. We can move atomically a location to another site. We represent mobile agents by locations. Agents can contain mobile sub-agents represented by nested locations. Agents move as a whole with all their current sub-agents, thereby locations have a dynamic tree structure. Location names are treated as first class values with lexical scopes, as is the case for channel names; the scope of every name may extend over several locations, and may be dynamically extended as the result of message passing or agent mobility. A location controls its own moves, and can move towards another location by providing the name of the target location, which would typically be communicated only to selected processes. In this section, we will present firstly the syntax of this calculus, and then some examples will be showed.

2.1 The Syntax of the Distributed Join Calculus

A system specified in the DJC is seen as a set of configurations. These configurations evolve in time and can interact. A specification in a DJC is composed of a set of terms. Terms in the DJC can be processes, definitions or configurations.

The grammar for processes is as follows:

P::=

0 // the inert process

| *P*|*P'* // Parallel Composition of two processes *P* and *P'*

| *x*<$\hat{y}$> // an emission of an asynchronous polyadic message $\hat{y}$ on the channel *x*

| *Def D in P* // a definition *D* in a process *P*, see the grammar of Definition

| *go a*; *P* // a request to migrate the current location toward the location *a*, then to execute the process *P* in the destination location (which is *a*)

The grammar of definitions is as follows:

D::=

T // the empty definition

| *D, D'* // Composition of two definitions D and D'

| *J*▷*P* // a reaction rule: this rule will wait for the arrival of a set of messages that match messages defined in the join pattern J, then the process P will be executed

| *a[D:P]* // declares a new location which name is *a*. In this location, we have a definition D and a running process P.

A pattern J is defined as follows:

J::=

x<$\hat{y}$> // waiting for a message $\hat{y}$ on the channel *x*. The pattern J will be satisfied when the message arrives.

| *J*|*J'* // Synchronization of two patterns J and J'. The pattern J|J' is satisfied when the two patterns are satisfied.

Finally, configurations have the grammar:

S::=

Ω // the empty configuration.

| $S||S'$ // Composition of two configurations.

| $D \vdash^{\alpha} P$ //where α is a sequence of locations that represents a path.

2.2 The Execution in the Distributed Join Calculus

The execution in the DJC is done using the principle defined in the Distributed CHemical Abstract Machine (DCHAM). A system specified using the CHAM-model [18] is seen as a set of molecules that evolves in a set of solutions. In a solution, molecules can evolve and change when reaction conditions are satisfied. Formally a solution is written as: $R \vdash M$, where M is the set of molecules and R is a set of reaction rules. The molecules M represent a join calculus processes running in parallel, and R models the current reduction rules (join calculus definitions). When a reaction is done, the molecules M can change into M'. This reaction is modelled through a ***reduction step*** denoted:

$(R \vdash M) \rightarrow (R \vdash M')$

In the DJC, a configuration can be considered as a solution. The unique reaction rule is the join pattern matching: J▷P. This rule will change the processes P by instantiating the formal parameters in P which match with the received parameters in the asynchronous messages defined in J.

As an example, we have the solution:

$\phi \vdash$ ready(laser), job(1), job(2)

In this solution, we have a printer with the name laser which is *ready* to print (*ready(laser)* is a molecule), and we have two jobs 1 and 2 (*job(1), job(2)* are two others molecules). In the DJC concepts, we have three asynchronous emissions of messages: message *laser* sent on the channel *ready*, messages *1* and *2* sent on the channel *job*. This solution has not a reaction rules. We can also write the solution as:

$\phi \vdash$ ready(laser)| job(1), job(2)

We can add to this solution the reaction rule:

D= ready(printer)|job(file) ▷ printer (file)

This reaction rule means that if we have some *printer* ready and some *job* which name is *file*, so we can send this *file* to this *printer* and print it. Now if we add this reaction rule to our solution, we will have the new solution:

D ⊢ ready(laser)| job(1), job(2)

We see that **(** *ready(laser)|job(1)* **)** in the set of molecules matches with **(** *ready(printer)|job(file)* **)** in the definition *D*, so the formal names *printer* and *file* in the definition *D* can be instantiated to the two parameters : *laser*, and *1*, and the join pattern (*ready(printer)|job(file)* ▷ *printer (file)*) can be satisfied. In this case, the solution can evolve. A reduction step can be applied to send the job *1* to the printer which name is *laser*.

ready(printer)|job(file) ▷ printer (file) ├ ready(laser)| job(1), job(2)

→

ready(printer)|job(file) ▷ printer (file) ├ laser(1), job(2

Formally, we have applied a join reduction rule (**Reaction-rule**), this rule can be written as:

Reaction-rule: $(J \triangleright P \vdash J_{\delta rv}, \ldots) \rightarrow (J \triangleright P \vdash P_{\delta rv}, \ldots)$

where δ_{rv} is a substitution on received variables (parameters of reception messages in joins). If we review the above example, the received variables are *printer* and *file* which will be substituted to the two variables *laser* and *1*. So we have the substitutions:

$J_{\delta rv}$ = (ready(printer)|job(file))$_{\delta rv}$ = ready(laser)| job(1)

and

$P_{\delta rv}$ = (printer (file))$_{\delta rv}$ = laser(1)

Beside the join pattern matching reaction rule that can be used to create reactions rules in solutions, the two following structural rule are also defined:

$(\phi \vdash \text{Def D in P}) \rightarrow (D \vdash P)$

$(D \vdash P) \rightarrow (\phi \vdash \text{Def D in P})$

These two rules show the reflexive property of the CHAM. We speak about a Reflexive CHAM (RCHAM). These two rules can be combined in one single rule

str-def-rule: $(\phi \vdash \text{Def D in P}) \leftrightarrow (D \vdash P)$

This structural definition rule shows that from a solution where we have only a process (*Def D in P*) and no reaction rules, we can have a solution where the joins defined in *D* will be considered as reactions rules for a solution that contained the process *P*.

In a DCHAM (Distributed CHAM), we have a set of solutions. We use the operator || to model the composition of several solutions. Solutions can evolve each one separately and can interact. Solutions interact using the communication reduction rule (COMM-rule):

COMM-rule: $((\phi \vdash x\langle\hat{y}\rangle) \| (D \vdash)) \rightarrow (D \vdash x\langle\hat{y}\rangle)$

In this rule, *x* must be a name of a port defined in the definition *D*. This rule means that a message emitted in a solution on a port defined in another solution can be sent to that solution where the port was defined.

To consider localities, solutions can be labelled with name of locations where they are residing. For example the solutions: $D \vdash^{\alpha} P$ is labelled with the path α of locations. This path α of locations can be only one location or a sequence of nested location. For example, in a printing system, we can distinguish three sub-systems: A machine user *u* from which printing requests are sent:

$\vdash^{u}$ job<1>

a machine server *s* that hosts the printing spooler :

D=ready(printer)|job(file) ▷ printer (file)

$D \vdash^{s}$

and finally, a laser printer *p* which is registered to in the server as ready and which hosts the printing code *P*

$laser(f) \triangleright P \vdash^{p} ready\langle laser\rangle$

The whole distributed system will be modelled as:

$\vdash^{u} job\langle 1\rangle \parallel ready(printer) | job(file) \triangleright printer(file) \vdash^{s} \parallel laser(f) \triangleright P \vdash^{p} ready\langle laser\rangle$

The above communication rule can be generalized to consider localities:

COMM-rule: $((\phi \vdash^{\alpha} x\langle\hat{y}\rangle) \| (D \vdash^{\beta})) \rightarrow ((\phi \vdash^{\alpha}) \| (D \vdash^{\beta} x\langle\hat{y}\rangle)$

Now, we can apply the two rules COMM-rule and Reaction-rule to make an execution in the above distributed system, as following:

$\vdash^{u} job\langle 1\rangle \parallel D \vdash^{s} \quad \parallel laser(f) \triangleright P \vdash^{p} ready\langle laser\rangle$

$\rightarrow^{\text{COMM-Rule between u and s}} \vdash^{u} \quad \parallel D \vdash^{s} job\langle 1\rangle \quad \parallel laser(f) \triangleright P \vdash^{p} ready\langle laser\rangle$

$\rightarrow^{\text{COMM-Rule between p and s}} \vdash^{u} \quad \parallel D \vdash^{s} job\langle 1\rangle, ready\langle laser\rangle \parallel laser(f) \triangleright P \vdash^{p}$

$\rightarrow^{\text{Reaction-Rule in s}} \vdash^{u} \quad \parallel D \vdash^{s} laser(1) \quad \parallel laser(f) \triangleright P \vdash^{p}$

$\rightarrow^{\text{COMM-Rule between s and p}} \vdash^{u} \quad \parallel D \vdash^{s} \quad \parallel laser(f) \triangleright P \vdash^{p} laser(1)$

In this execution, the first step sends the job *1* from the user machine to the server machine, and the second step sends the name of the ready printer *laser* to the server. The first step is realized on the server machine and will affect the job *1* to the printer *laser*. The final step resends the job *1* toward the printer laser which will print it.

The definition $a[D:P]$ which defines a new location named *a*, where a definition *D* is declared and a process *P* is running, corresponds to the solution $D \vdash^{\alpha a} P$, where α is the path of the local definitions in *D*. Two structural rules are defined that represent folding and unfolding of path of locations:

LOC-rules: $a[D:P] \vdash^{\alpha} \rightarrow D \vdash^{\alpha a} P$

$D \vdash^{\alpha a} P \rightarrow a[D:P] \vdash^{\alpha}$

These two rules can be combined in one reflexive rule: $a[D:P] \vdash^{\alpha} \leftrightarrow D \vdash^{\alpha a} P$

Finally migration of process is realized with an instruction *go a*; *P* which will transfer the current location to the target location named *a*, where the process P will be executed. The execution of the migration is done by the reaction rule:

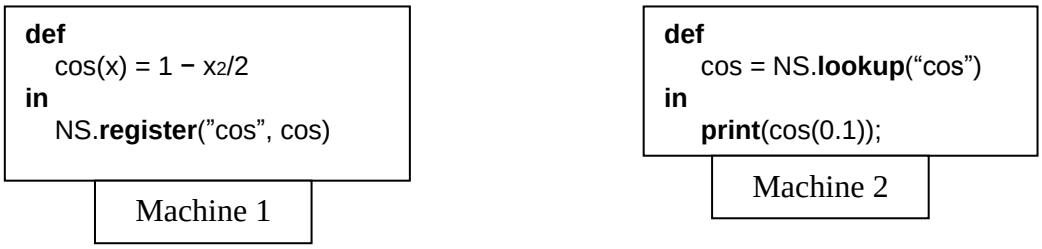
GO-rules: $\vdash^{\alpha b} go\ a; P \parallel \vdash^{\beta a} \rightarrow \vdash^{\alpha b} \parallel \vdash^{\beta a} P$

This rule shows the migration of the process *P* from the solution labelled with the location b to the solution labelled with the location *a*.

2.3 Examples of specifications in DJC

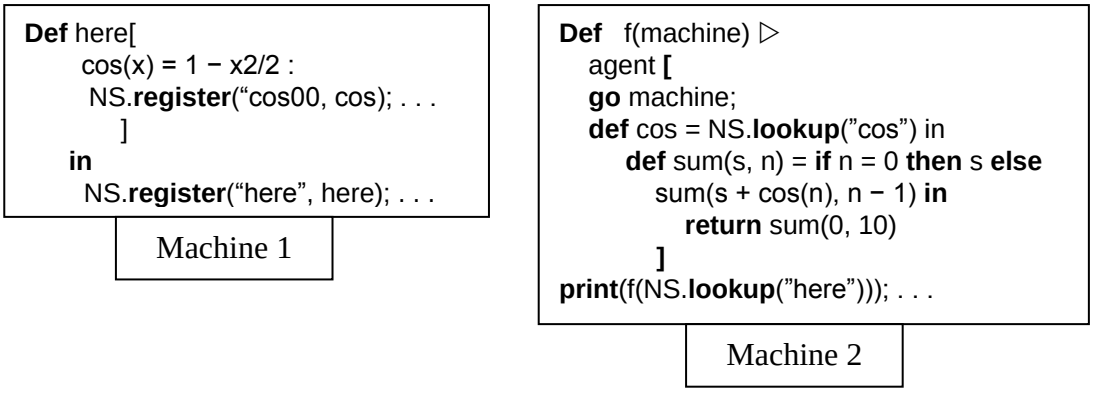
In the DJC, **Locality** is **explicitly** controlled in the language; this locality can be adjusted using migration. In contrast, resources such as definitions and processes are not silently relocated or replicated by the system. As an example in the implementation JoCaml [16], programs being run on different machines do not initially share any channel name; therefore, they would normally not be able to interact with one another. To bootstrap a distributed computation, it is necessary to exchange a few names; this is achieved using a built-in library called the name server (NS). Once this is done, these names can be used to communicate some more names and to build more complex communication patterns. The interface of the name server consists of two functions to register and look up. The first example shows the use of a remote function (**cos**) using a remote call. The second example shows how to use this function when this function is defined in an explicit location and the using program of this function is an agent which is located on a remote machine and which can migrate to the machine where **cos** is defined.

Example 1:



In this first example, on the machine 1, a process defines a local name **cos** and registers it to the name server NS. On the machine 2, a second process obtains the name **cos** and does a remote call.

Example 2:



In this second example, on machine 1, the process explicitly, defines a named location. This location wraps the function definition (cos). The process exports the location name under the key “here”. On machine 2, we define a function **f** with one formal parameter **machine**. This function defines a new location **agent**. This agent will start by migrate to the location named **machine** (as written in the first instruction: **go machine**). On the location machine, the agent will execute a program sum which will use the function cos defined on the location

machine. Effectively, the execution will start by the instruction: **print**(f(NS.**lookup**("here"))) which instantiates the formal name **machine** to an effective name : **here**.

3 Towards Dynamic Petri Nets

In this section, we will present the ***Dynamic Petri Nets*** formalism. This formalism is one of the first one proposed to model mobile systems. The Dynamic Petri Nets extends Petri Nets formalism to model dynamic reconfiguration of the structure of the net. This reconfiguration of the structure is not possible in Classical Petri Nets. This reconfiguration of the structure will enhance the expressiveness of Petri Nets, and of course allow the modelling of mobility. The Dynamic Petri nets are also inspired from the Join Calculus. To present Dynamic Petri Nets, we will present initially ***Petri nets*** and ***mobile Petri nets*** which are the two basic formalisms for DPN.

3.1 Petri Nets

Definition 3.1.1.

Informally, A Petri net is a bipartite graph, where nodes are composed of two disjoint sets, a set of ***places*** and a set of ***transitions***. ***Arcs*** are used to link transitions and places in the graph. Transitions can be linked only to places and places can be linked only to transitions. A marked Petri net is a Petri net where places can store temporally some ***tokens***. The tokens stored in a place represent the ***marking*** of this place. A bounded place could not contain an infinite number of tokens. Arcs can be labelled with ***weights***. If all arcs are weighted 1, so the ***Petri net*** is said to be ordinary. A place is said to be an ***input place*** of a transition, if there is an arc from this place to this transition. A place is said to be an ***output place*** of a transition, if there is an arc from this transition to this place. The dynamic of the Petri net is created by the ***firings*** of transitions in the net. A transition can be fired if some conditions are insured. When a transition is fired, the marking of its input places and the markings of its output places are updated. We say that the firing of a transition will update the marking of the net. This dynamic of Petri net can be used to model a system where there are some actions that can be executed concurrently, or in parallel.

Definition 3.1.2.

Formally, a Petri net N can be seen as a 6-uplets $N=(P, T, A, W, B, M_0)$, where :

P: is a finite set of places, $P=\{p_1, ..., p_n\}$,

T: is a finite set of transitions, $T=\{t_1, ..., t_m\}$,

A: is a finite set of arcs, $A \subseteq (P \times T) \cup (T \times P)$,

W: is a function of weights, $W:A \rightarrow \mathcal{N}$. By $\mathcal{N}$, we denote the natural numbers set. W maps each arc to its weight,

B: is a function of bounds, $B:P \rightarrow \mathcal{N}$. B maps each place to its bound,

M_0: is a function that defines the initial marking of each place, $M_0: P \rightarrow \mathcal{N}$.

Dynamic of a Petri net:

For each transition $t \in T$, we denote by °t, the set of the input places of t, and by t° the set of the output places of t.

From some marking M of the Petri net N, the transition t can be fired (and we say that t is fireable) iff:

$\forall p \in {}^\circ t$: $M(p) \geq W(p,t)$

$\forall p \in t^\circ$: $M(p) < B(t,p)$

if these two conditions are satisfied, t can be fired, and so the marking M will be updated to a new marking M', such that:

$\forall p \in {}^\circ t$: $M'(p) = M(p) - W(p,t)$

$\forall p \in t^\circ$: $M'(p) = M(p) + W(t,p)$

and we say that the marking M' is reachable from M when t is fired. This can be denoted as: $M \rightarrow^{t} M'$

With these few concepts, one can model many concurrent systems, where processes must be synchronized on some resources. This model can also be used to verify many properties of such systems. Reachability, liveness, soundness, reversibility … and other properties can all be verified using this classical version of Petri nets. Currently, there are many maturated techniques that the developer can apply to verify such properties in a system modeled using Petri Nets. However, the use of this classical version to model some sophisticated systems (like mobile, or reconfigurable systems) don't allow to the developer to catch all the properties of this kind of system. The expressiveness of this model is so poor to specify some new and inherent qualities in mobile systems. This poorness in expressivity oblige the developer using classical Petri net, to have complex Petri net model that are difficult to understand and so to verify if they represent really the needed system.

One of the important propositions to model mobility using Petri net is to extend this model in order to catch the most important property of these systems, which is reconfigurability (or dynamicity) of the system's structure. The easiest idea is to consider that the dynamic structure of a mobile system can be directly coded in a Petri net where the structure of the graph changes over time. The changing of a mobile system's structure is specified directly and explicitly through the changing of the Petri net's graph. In Mobile Petri net, the idea was to allow the creation of some new output places when some transition is fired. In dynamic Petri nets, more ability will be offered, and the firing of a transition will add a new net to the first one. The following paragraphs present in more details these two formalisms, which have inspired our works in this thesis. The presentation of these two formalisms will use a new syntax not commonly used in the definition of Petri nets. This syntax is inspired from the Join-calculus. The reason of this syntax is due to the principal that authors of Mobile nets have applied the idea of mobility in Join-calculus, to extend Petri net, with mobility. In the following definitions, places are considered as names like those used in the terms of the Join-calculus. Firstly, we will recall the definition of Petri nets using the syntax of the Join Calculus. The presentation of Mobile Petri nets and Dynamic Petri nets will be done in the next two subsections:

Definition 3.1.3. (multi-set)

Given a set *X*, a multi set over *X* is a function *m*, defined as: $m: X \rightarrow \omega \cup \{\omega\}$. The set of all multi sets over *X* is denoted by M_X.

Let : $dom(m) = \{x \in X, m(x) > 0\}$,

A multi-set over *X* is said to be empty if for all $x \in X, m(x) = 0$,

We define two sets:

$$M_X^{post} = \{m \in X \mid dom(x) \; is \; finite\}$$

$$M_X^{pre} = \{m \in M_X^{post} \mid m \; is \; not \; empty \; and \; \forall x \in X, m(x) \in \omega\}$$

We have:

$$(m \oplus m')(x) = m(x) + m'(x)$$

$$(m \setminus m')(x) = m(x) - m'(x) \; iff \; m(x) \geq m'(x)$$

$$(m \setminus m')(x) = 0 \; \; iff \; m(x) < m'(x)$$

Definition 3.1.3. (Petri Net)

let *X* be a set of names, these names will be used to indicate names of places in a net.

let $Y \subseteq X$, *Y* can represent the names of the places of a net *N*,

let $T \subseteq M_X^{pre} \times M_X^{post}$, *T* can represent a set of transitions of the net *N*,

let $m \in M_X$, *m* can represent an initial marking of the net *N*,

Where $N = (\upsilon Y)(m, T)$, υY denotes that the set *Y* represents fresh names; these names are restricted in this net.

N is a Petri net iff all names occurring in its initial marking and in their transitions are contained in the set of places. Formally: $dom(m) \cup \bigcup_{(c,p) \in T} (dom(c) \cup dom(p)) \subseteq Y$.

For a name *x*, we denote by *m(x)* the marking of the place *x*.

A transition *t* is denoted *(c,p)* or $c \triangleright p$. In this transition, *c* denotes the tokens to be consumed when *t* is fired, and *p* denotes the set of tokens to be produced.

The transition $t = c \triangleright p$ can be fired in a marking *m* iff $c \subseteq m$. When *t* is fired, the marking *m* of the net will change to *m'*, $m' = (m \setminus c) \oplus p$

3.2 Mobile Petri Nets

Mobile Petri nets are an extension of Colored Petri nets. In Mobile Petri nets, names of places can be tokens that mark some other places. Mobile Petri nets have a dynamic structure in such way that the names of output places of a transition can be defined dynamically when this transition is fired. It is clear, that this idea is inspired from π-calculus, where the gates can be sent from one process to another, and then, the received gate can be used. This idea is used so in Mobile Petri nets to make dynamically links from one transition to some place, if this last one marks one of the input places of the transition. In this section, we present the formal basics necessary to the definition of Mobile Petri nets.

We extend the definition of multi-sets defined previously on one set *X*, to two sets: *X*, and *Y*:

Given two sets *X* and *Y*, the multi-sets over $X \times Y$ is $M_{X,Y} = X \rightarrow (Y \rightarrow (\omega \cup \{\omega\}))$.

An element $m \in M_{X,Y}$ can be interpreted as a marking. If for some *x, y* we have $m(x)(y) > 0$, this means that the marking of the place *x* is *y*. In this interpretation, there is no constraint on the name *y*. This means that this name *y* can denote also a name of a place.

Let $dom(m) = \{(x, y) / m(x)(y) > 0\}$, this set can be interpreted as the set of marked places (not empty) in a marking *m* of a net.

$M_{X,Y}^{post} = \{m \in M_{X,Y} / dom(m) \text{ is finite}\}$, The set $M_{X,Y}^{post}$ can be interpreted as the set of **finite** markings defined on the tow sets *X*, and *Y*. Where *X* represents the set of places, and *Y* represents the set of tokens that can mark the places defined in the set *X*.

and $M_{X,Y}^{pre} = \{m \in M_{X,Y}^{post} / m \text{ is not empty } \wedge \ \forall x \in X, \forall y \in Y \ m(x)(y) \in \omega\}$.

The operator $\oplus$ is defined now as: $(m \oplus m')(x)(y) = m(x)(y) + m'(x)(y)$. The operator \ is defined now as: $(m \setminus m')(x)(y) = m(x)(y) \setminus m'(x)(y)$.

The set of colours denoted *C* can be defined as:

$C = \{(x_1, ..., x_n) / n \geq 0 \quad and \ x_i \in X, \ i = 1, .., n\}$, in this definition of colours there is no constraint on names that can appear in a tuple. These names can denote also places.

We denote by $\overline{x}$ the finite tuple of names $(x_1, ..., x_n)$. The length of the tuple $(x_1, ..., x_n)$ is defined as $|(x_1, ..., x_n)|$=*n*. The element x_i is denoted by: $\pi_i(x_1, ..., x_n)$.

A substitution is a partial function defined over *X* as:

$$x\rho = \begin{cases} y & if \ (x,y) \in \rho \\ x & otherwise \end{cases}$$

This means that the name *y* can be substituted by the name *x*.

A substitution on a tuple is defined as: $\overline{x}\rho = (x_1, ..., x_n)\rho = (x_1\rho, ..., x_n\rho)$

Given $m \in M_{X,C}$, the substitution on all names defined in a marking *m* is defined as:

$$(m\rho)(x)(\overline{y}) = \sum_{v\rho = x \text{ and } \overline{z}\rho = \overline{y}} m(v)(\overline{z})$$

In this substitution, the name *x* (which can represent the name of a place) is substituted by the name *v*. The tuple $\overline{y}$ (a tuple of names that can represent the marking of the place *x*) is substituted by the tuple $\overline{z}$.

The substitution can be restricted only on the names occurring in the marking of a place. In this case, the substitution is denoted and defined as:

$$(m_b\rho)(x)(\overline{y}) = \sum_{\overline{z}\rho = \overline{y}} m(x)(\overline{z})$$

A transition has a **preset** (the set of preconditions necessary to fire this transition) and a **postset** (the new markings of the output places of this transition). To define the Dynamic Net and the firing rules for transitions, we define the functions of ***free*** names (*fn*), ***bounded*** names (*bn*), ***free names in pattern*** (fn_P), ***free names in marking*** (fn_M), ***bounded names in pattern*** (bn_P), and ***bounded names in marking*** (bn_M). We will need to define **free** and **bound** names in a transition. These free and bound names differ between the preset (a pattern) to the postset (a marking). These functions are defined as bellow:

$fn_p(m) = \{x \mid \exists \overline{y}, \ m(x)(\overline{y}) > 0\}$, this means that the free names in a preset are the set of places with a marking>0.

$bn_p(m) = \{x \mid \exists z, \exists \overline{y}, \exists i, \ \pi_i(\overline{y}) = x \ and \ m(z)(\overline{y}) > 0\}$, this means that a bound name is a name that occurs in the marking of a place *z*.

$fn_M(m) = fn_p(m) \cup bn_p(m)$;

$bn_M(m) = \phi$; which means that in a postset all names are free.

Now we define free and bound names in a transition. In a transition ($c \triangleright p$), we have:

$fn(c \triangleright p) = fn_p(c) \cup (fn_M(p) \setminus bn_p(c))$, this means that the set of free names in a transition are all the **free names defined in its preset** added to the **set a free names defined in its postset** which must **not be bounded in the preset**.

Definition 3.2.1. (Mobile Petri Net)

Let *C* be a set of colours, *X* a set of names.

Let $N = (\upsilon Y)(m, T)$, where: $Y \subseteq X$ represents the set of places, $m \in M_{X,C}$ is the initial marking, and $T \subseteq M^{pre}_{X,C} \times M^{post}_{X,C}$ is the set of transitions.

N is a mobile net iff: $fn_M(T) \cup fn(m) \subseteq Y$, this means that the free names defined in the initial marking and the free names defined in all the transitions of *N* must be only names of places declared previously in *Y*. This condition will insure that all new places that can be added to the net when a transition is fired are previously declared in the net *N*. So this condition will insure that the set of places are finite, even connections can be created dynamically between transitions and output places.

Dynamic of mobile Petri nets:

The transition $t = c \triangleright p$ is enabled at a marking *m*, iff there is as substitution ρ that can satisfy the conditions to fire *t*. The satisfaction of *t* is insured if ρ substitutes each set of names in a marking of a place defined in the preset *c* with the marking of this place in the marking *m*. So the *t* is enabled iff: $c_b\rho \subseteq m$. The substitution is restricted to the names occurring as markings in m. The firing of *t* will update the marking *m* to a new marking *m'* defined as: $m' = (m \setminus c_b\rho) \oplus p\rho$. The substitution applied on the postset *p* concerns as names occurring in marking as names of places containing these markings. This substitution will so make that output places of *t* depends on the preset of *t*. The set of output places of *t* is not rigid but it is dynamic.

Example of mobile Petri nets:

As an example, let consider the net $N = (\upsilon Y)(m, T)$ where:

$Y = \{p_1, p_2, p_3, p_4, p_5\}$, $T = \{t_1, t_2\}$,

$t_1 = \{p_1(x, a, p), p_2(y, z)\} \triangleright \{p(x, y)\}$, $t_2 = \{p_2(p, p')\} \triangleright \{p(1, 2), p'(a, b), p_3(6)\}$

Suppose that the initial marking is $m = \{p_1(1, a, p_5), p_2(p_4, p_5)\}$. From this initial marking, the transitions t_1 is enabled with the substitution: $\rho_1 = \{(1, x), (a, a), (p_5, p), (p_4, y), (p_5, z)\}$. The firing of t_1 will update the marking to: $m' = \{p_5(1, p_4)\}$. From the initial marking, the transitions t_2 is enabled with the substitution: $\rho_2 = \{(p_4, p), (p_5, p')\}$. The firing of t_2 will update the marking to: $m' = \{p_1(1, a, p_5), p_4(1, 2), p_5(a, b), p_3(6)\}$. The figure II.1 shows this example.

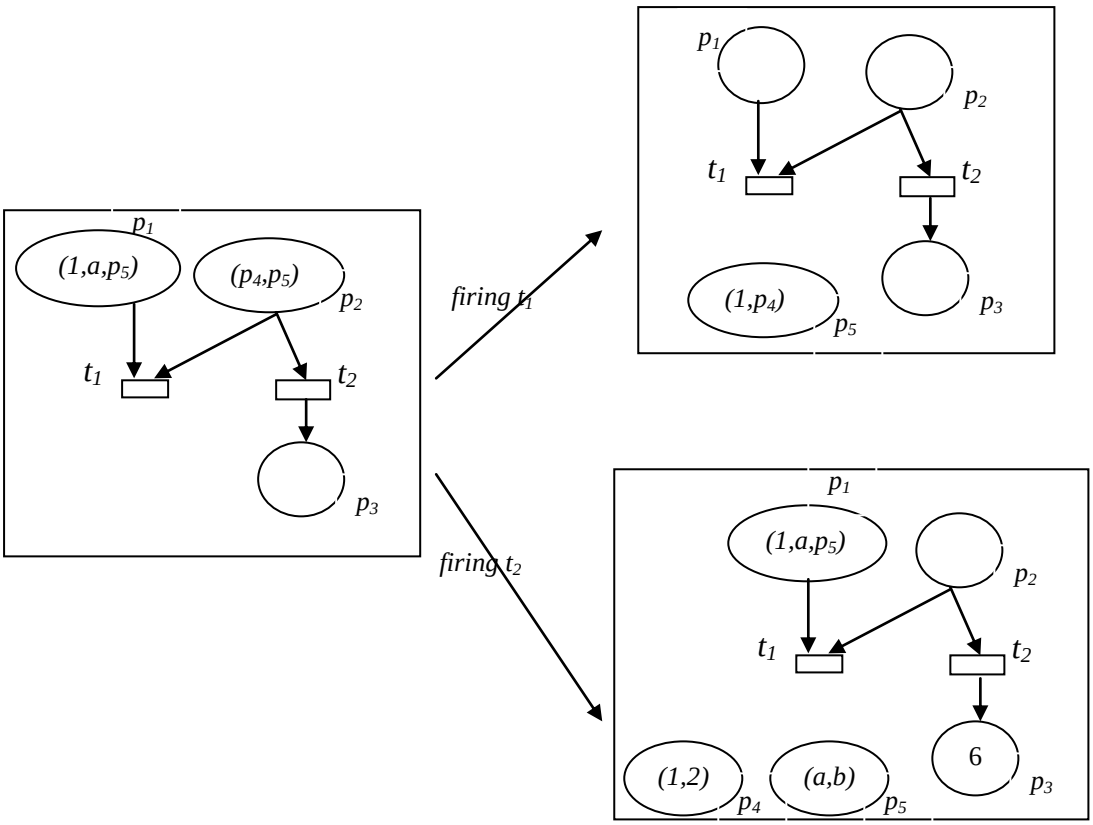


Figure II.1. A graphical representation of mobile nets' dynamic

3.3 Dynamic Petri Nets

In the mobile Petri nets version, the dynamic of the structure of the net is so poor. We allow only the change of output connections of some transitions. So there is not creation of new places or new transitions. The only allowable dynamic consists of the modification of output connections. In the Dynamic Petri nets, the idea is to extend mobile Petri nets with the possibility to add not only output connections to some transitions, but also some whole transitions to the net when some transitions are fired. So, in Dynamic Petri nets we allow the creation of whole new nets when some transitions are fired.

We define the set DN as the least set that satisfies the following equations:

$$\mathcal{N} = \{(\upsilon Y)(T, m) \mid$$

$$Y \subseteq X,$$

$$T \subseteq \{c \triangleright N \mid c \in M_{X,C}^{pre}, N \in \mathcal{N}\},$$

$$m \in M_{X,C}\}$$

In this definition, the postset of a transition is a new net. This new net will be added to the original net where the transition exists.

Free and Bound Names:

Let $(\upsilon Y)(T, m) \in DN$. Y is the set of places; T is the set of transitions and m the initial marking. Here after, we show the free names in a transition, in a set of transition, and finally in a net:

$fn(c \triangleright N) = fn_p(c) \cup (fn(N) \setminus bn_p(c))$;

$fn(T) = \bigcup_{t \in T} fn(t)$;

$fn((\upsilon Y)(T, m)) = (fn(T)) \cup fn_M(m)) \setminus Y$;

Substitutions:

A a substitution ρ on an element *m* can be defined on all names in *m* (denoted ($m\rho$)) (in case where *m* is a postset) or only on names in tokens (denoted ($m_b\rho$)) (in case where *m* is a pattern (i.e a preset)):

Let $t = c \triangleright N \text{ and } bn_p(c) \cap n(\rho) = \phi;$ where $n\rho = \cup_{(x,y)\in\rho}\{x, y\}$. This means that when the substitution ρ is applied there will not be confusion between new names appearing after the substitution and the bounded names defined in *c*. This is necessary to avoid that the new names due to the substitution will be captured by a binders defined in the preset *c*.

A substitution on transitions will be defined as:

$$t\rho = c\rho \triangleright N\rho$$

and $T\rho = \{t\rho / t \in T\}$

Let $N = (\upsilon Y)(T, m) \text{ and } Y \cap n(\rho) = \phi;$ the substitution on nets will be defined as:

$$N\rho = (\upsilon Y)(T\rho, m\rho)$$

Let $N_1 = (\upsilon Y_1)(T_1, m_1) \text{ and } N_2 = (\upsilon Y_2)(T_2, m_2)$, to be two nets;

If $Y_1 \cap Y_2 = \phi,\ fn(N_1) \cap Y_2 = \phi,$ and $fn(N_2) \cap Y_1 = \phi$, we extend the operation $\oplus$ to nets, and we will have:

$$N_1 \oplus N_2 = (\upsilon Y_1 \cup Y_2)(T_1 \cup T_2,\ m_1 \oplus m_2)$$

in case where: $Y_1 \cap Y_2 \neq \phi,\ fn(N_1) \cap Y_2 \neq \phi,$ or $fn(N_2) \cap Y_1 \neq \phi$, we can apply some alpha-conversion to avoid confusion between names defined in different sets.

If $Y_1 \cap Y_2 = \phi$ then we have $(\upsilon Y_1)N_2 = (\upsilon Y_1 \cup Y_2)(T_2, m_2)$

Definition 3.3.1. (Dynamic Nets)

A Dynamic Net *N* is an element of *DN* which is closed: $fn(N) = \phi$.

Firing Rules:

Let $N_1 = (\upsilon Y_1)(T_1, m_1)$ and $t = c \triangleright N$ be a transition in T_1:

t is enabled in N_1 iff there exists $\rho \subseteq bn(c) \times X$ such that: $c_b\rho \subseteq m_1$;

The firing of *t* in N_1 with substitution ρ produces the new net: $N_2 = (\upsilon Y_1)((T_1, m_1 / c_b\rho) \oplus N\rho)$. We denote this as: $N_1 \rightarrow^{t_1} N_2$

Example of a Dynamic net:

As an example, let consider the following dynamic net:

$$
\begin{array}{l}
(\upsilon\{A, B\})(\\
\quad \{A(X) \triangleright \\
\qquad (\upsilon\{Y\})(\\
\qquad\quad \{X(W), Y(Z)\} \triangleright W(Z), \\
\qquad\quad \{A(Y), Y(B)\} \\
\qquad), \\
\quad \{A(A), B(B)\} \\
)
\end{array}
$$

In this example, we have an external net with two places, one transition, and an initial marking. The two places are: {*A*, *B*}, so these two names are bounded in all the net. The initial marking is {*A(A), B(B)*}, the place *A* is marked with the name *A* which is the name of this place itself, and the place *B* is marked with the name *B*, which is also the name of the place itself. The transition has as preset: *A(X)*, so the firing of the transition requires the existing of some name *X* in the place *A*. The postset of this transition is a new net, so the firing of this transition will add a new net. The new net, has one place {*Y*}, one transition (internal), and its initial marking is *A(Y), Y(B)*. The name *Y* is bounded in the internal net. The internal transition has as preset: {*X(W), Y(Z)*}, so the firing of this transition requires the existing of a name *W* in the place *X*, and a name *Z* in the place *Y*. The postset of the internal transition consists to add the name *Z* in the place *W*. We see that all names of places are bounded in this net. So this is a correct dynamic net.

The external transition can be fired with the substitution {*(A,X)*}, where *A* is substituted to *X*. After the firing of the external transition, we will have a new net (the internal net, where *X* is replaced by *A*):

$$
\begin{array}{l}
(\upsilon\{Y\})(\\
\quad \{A(W), Y(Z)\} \triangleright W(Z), \\
\quad \{A(Y), Y(B)\} \\
\quad)
\end{array}
$$

In this new net, the transition can be fired using the substitution {(*Y,W*), (*B,Z*)}. Once fired, we will have a new marking: *Y(B)*. So after the firing of the two transitions, the new marking of the net will be {*B(B), Y(B)*}.

3.4 Verification of Dynamic Nets

The use of Petri nets in specification of systems finds its advantages in the sets of verification and analysis techniques. The poorness of classical Petri nets in modelling levels offers an important gain in the analysis levels. When one uses classical Petri nets, we can find many analysis techniques: the reachability graph, algebraic techniques, and reductions techniques. The use of high order Petri nets makes the analysis level more complex, and sometimes impossible for some properties. One idea is to unfold (transform from high level to low level) models of high order Petri nets into classical ones then to do analysis on these last ones. The case of Coloured Petri nets is an example, where models can be unfolded into

classical Petri nets. In this sense, the authors of Mobile and Dynamic nets have proposed in [20] an encoding of Mobile nets into classical nets, and then an encoding of Dynamic nets into mobile nets. Using these encoding, the designer can specify its system into dynamic nets then use the encoding process to unfold its models into low level ones that can be analysed using known techniques for classical Petri nets. This encoding will be presented in the appendix A in this thesis.

4 Conclusion

In this chapter, we have presented some formal methods proposed to specify mobile systems. We have considered that works on formal methods for mobility can be divided into two axes: Process algebra, and Petri nets. In the process algebra, most works are based on the classical π-calculus. In this chapter, we have not presented all calculi, but we have presented only the Join-calculus and its distributed version. The join calculus is an extension of the π-calculus, where the concept of localities is explicit and where migration of processes (or agent) is realized using a *go* instruction. The join calculus has been used to model some mobile systems and represents the formal background of the programming language Jo-Caml [16].

In this thesis, we are interested to the use of Petri nets in the modelling of mobility. In this chapter, we have presented the definition of classical Petri nets. This model is a well known model in concurrency domain. It can be used to specify distributed systems, and to verify many of their properties. The use of Petri nets (with their basic definition) to model mobility is not easy. The designer is obliged to code mobility in a rigid model that does not support this property. One good idea was to extend Petri nets with the possibility to change their structure during their execution. Once this is possible, the modelling of mobile system (where structure changes over time) will be direct and easier in a high level Petri net (with a dynamic structure over time). During our study, we have found many extension of Petri net that were proposed to deal with mobility. We have seen that the richest extensions were Mobile Petri nets and Dynamic Petri nets. Mobile Petri nets can be seen as Petri net translation of the π-calculus. In mobile nets, the input of a transition defines its output. The output connections of a transition are not fixe, but they can be defined at runtime. This can be used to specify systems, where connections between their components change over time, during the execution of this system. In Dynamic nets, the structure can change in a more free way. In this last formalism, new nets can be created during the firing of some transitions. These nets can model new components that are downloaded by the local system. So using Dynamic nets, one can specify in a direct and an implicit way, downloading of new components, which is one of the application fields in mobile code systems.

These two extensions of Petri nets offer, probably, to the designer two new tools based on Petri nets that can be used to model, in an easy way, systems where structure changes at runtime. However the powerful of analysis techniques defined for Petri nets is lost. The authors of Mobile and Dynamic nets have proposed an encoding of these two formalisms into low level Petri nets. This encoding can be used to unfold high level model into low level formalism then to analyze this last one.

Even the powerful of these two formalisms, that we consider the most advanced formalisms in the field of *Petri nets for mobility*, we have concluded that these two

formalisms have many limits. It is clear that the idea in Dynamic Petri nets is to offer the possibility to add new transitions to a net, when some other transitions are fired. The idea is always to add new component to a net. This does not reflect the reality, where the systems change in a reduction way (Components, which disappear). The idea of elimination of places, connections, or transitions in a free way is not possible in mobile nets neither in dynamic nets. In mobile nets, connection can disappear but only from output of transitions. The input connection must not be eliminated. The process of adding places in mobile nets or transitions in dynamic nets is also constrained and must respect some conditions. In mobile nets, it is clear that it is not possible to add input places to transitions, so it is not possible to change the input connections. In dynamic nets, the transitions which can be added must not change the existing ones. In this sense, it is always impossible to change an existing transition.

Our principal idea in this thesis is to offer more flexibility to Petri nets. We propose a more adaptable formalism. We want to extend Petri nets towards Flexible Petri nets. In the Flexible Petri nets, basic components which are: places, transitions, and connections can be added or deleted in a free way and at run time. This idea must offer the most flexible formalism to model mobile systems and in a large way all reconfigurable systems. As the dynamic Petri nets are the richest extension of Petri nets that we have found, we consider that the first encoding of Flexible nets can be done into the dynamic Petri nets. The next chapter will present in more details the Flexible nets formalism, will present the idea to encode this formalism into dynamic Petri nets, then it will give a proof for this encoding.

Chapter III:
Extended Petri Nets

1 Introduction

Formal methods proposed to deal with mobility are numerous. We have distinguished between those which extend processes calculi, and those which extend Petri nets. In many extensions of Petri nets, the idea was always to offer the ability to change the structure of the graph during the execution of the net. In these models, the designer will have a modeling tool where mobility can be modeled explicitly and more easily. However, once the formalism offers a more expressivity its analysis will become more and more complex. For this reason, many authors and so many formalisms make numerous constraints on the dynamicity in the net. During our study, we were interested to propose a new formalism that must offer more expressive power than those proposed currently. Our propositions have been published in some conferences and journals.

The first idea that has motivated our works was mobile code systems. In these systems, type of resources and their bindings play a central role in the migration process. Resources decide also the success or failure of the process. Proposed formal methods founded in the literature do not deal with these aspects and their problems. In our first work, we have proposed "Labelled Reconfigurable Nets" [42] extended to "Colored Reconfigurable Nets" in [45]. Our objective was to propose a graphical tool to model mobile code systems in an easy and intuitive way. In these works, we were interested to provide formalisms that model mobility explicitly. The mobility is modeled through the reconfiguration of the net's structure when **some transitions are fired**. When trying to offer this quality in a model, we have to deal with the problem of interpreting this reconfiguration formally. In [42] and [45], we have introduced specific transitions **"reconfigure transitions"**, which reconfigure the net when they are fired. The first drawback of this solution is that we must provide a specific treatment of these transitions when the model is analyzed. The second drawback is the use of rigid labels associated to the transitions which make the model proposed in [42] not a parameterized model. To cover these limits, we have adopted specific techniques to analyze these models. In [46], we have proposed an interpretation of reconfigurable labeled nets into a high order Maude (reconfigurable Maude). The idea was to extend Maude [37] with some reconfigure rewriting rules. These rules can represent the reconfigure transitions. Reconfigurable Maude can be used to simulate reconfigurable labeled nets. In another work [48], we have adopted a theoretical method. We have proposed some rules to translate reconfigurable transitions into classical transitions, so reconfigurable nets will be translated into Colored Petri nets.

In our latest works, we have proposed ***Flexible nets*** [47, 49] which will be presented in this chapter as our principal contribution in this thesis. In Flexible nets, we propose to deal with the reconfiguration of the net with another point of view. We introduce specific ***sorts*** in the model. These sorts will contain ***signed objects*** (places, transitions, and arcs). An ***internal operation*** will be defined in these sorts. This operation will add or delete objects based on the sign of these last ones. Types in the model can be constructed based on these sorts and other predefined types. Reconfiguration of the structure of the net is interpreted as an operation that manipulates this structure by manipulating their components which are signed objects. The presence of a positive object (resp. negative object) in some place can be a cause to add (resp. delete) this object to (resp. from) the structure of this net. The formalism proposed is called ***Flexible Nets*** and reflects the idea that the model has a dynamic structure. This structure can

be expanded, shrunken, or destroyed. Our most contribution in this thesis is the **definition of the formal model**: ***Flexible nets*** and the definition of its semantics, then the **proposition of the encoding** of this model into Dynamic nets. This encoding will be used to offer an idea to do the analysis. This **encoding will be also proved**.

To present our contribution, this chapter is organized as follows: The section two starts by presenting the *Labeled Reconfigurable Nets* (LRN) (as the first naïve idea). We present the motivation of this formalism, its formal definition and dynamic, than we show an example of use and we discuss its shortcomings. Section three presents a first extension of the LRN formalism to Colored Reconfigurable Nets (CRN) formalism. We will present the formal definition of RCN, its semantics, and an example of modeling; this section will be concluded by a discussion of shortcomings of this last formalism. Section four presents a more mature idea:"Flexible Nets", we present the formal definition, dynamics of this formalism and we show some examples. Section four discusses the proposed issues to analyze these models.

2 Labeled Reconfigurable Nets: A naïve idea

Labeled Reconfigurable Nets [42] attempts to provide a formal and graphical model for code mobility. We have extended Petri nets with *reconfigure labeled transitions* that when they are fired reconfigure the net. Mobility is modeled explicitly by the possibility of adding or deleting at runtime arcs, transitions and places. Modification in *reconfigure transition*'s label allows modeling different kinds of code mobility. Bindings to resources can be modeled by adding arcs between environments. It is clear that in this model created nets are in the same level of nets that create them. Creator and created nets can communicate. This model is more adequate for modeling mobile code systems.

Labeled reconfigurable nets are an extension of Petri nets. Informally, a labeled reconfigurable net is a set of *environments* (blocs of *units*). Connections between these environments and their contents can be modified during runtime. A *unit* is a specific Petri net. A unit can contain three kinds of transitions (a unique *start* transition: , a set of ordinary transitions: , and a set of reconfigure transitions:).

Preconditions and post-conditions to fire a start or an ordinary transition are the same that in Petri nets. Reconfigure transitions are labeled with labels that influence their firing. When a *reconfigure transition* is fired, a net N will be (re)moved from an environment E towards another environment E'. The net N, the environment E and E' are defined in the label associated to the transition. After firing a *reconfigure transition*, the structure of the labeled reconfigurable net will be updated (i.e some places, arcs, and transitions will be deleted or added). Here after we give our formal definitions of the concepts: unit, environment and labeled reconfigurable net. After the definition, we present the dynamic aspect of this model.

2.1 Formal Definition

Let $N_1, N_2, \ldots N_k$ be a set of nets.
for each i: 1, …, n : $N_i = (P_i, T_i, A_i)$, such that :

1. $P_i = \{p^i_1, p^i_2, \ldots, p^i_n)$ a finite set of places,
2. $T_i = ST_i \cup RT_i$
   - $ST_i = \{st^i_1, st^i_2, \ldots, st^i_m\}$ a finite set of standard (ordinary) transitions,
   - $RT_i = \{rt^i_1, rt^i_2, \ldots, rt^i_r\}$ a finite set (eventually empty) of "*reconfigure transitions*",

3. $A_i \subseteq P_i \times T_i \cup T_i \times P_i$.

Definition 2.1 (Unit): a unit UN is a net N_i that has a specific transition st^i_j denoted $start^i$. So $T_i=\{start^i\}\cup ST_i \cup RT_i$.

Définition 2.2 (Environment): an environment E is a quadruplet E=(GP, RP, U, A)

- $GP = \{gp_1, gp_2, \ldots, gp_s\}$ a finite set of specific places : "*guest places* ";
- $RP = \{rp_1, rp_2, \ldots, rp_s\}$ a finite set of specific places : "*resource places*";
- $U = \{ N_1, N_2, \ldots N_k\}$ a set of nets.
- $A \subseteq GP \times StrT \cup RP \times T$. Such that : $StrT=\{start^1, start^2, \ldots, start^k\}$ and $T=ST_1 \cup RT_1 \cup ST_2 \cup RT_2 \cup \ldots \cup ST_k \cup RT_k$

Definition 2.3 (Labeled reconfigurable net):

A labeled reconfigurable net LRN is a set of environments. $LRN=\{E_1, E_2, \ldots, E_p\}$ such that

- There exist at least one net N_i in LRN such that $RT_i \neq \varnothing$; (there is a reconfigurable transition)
- For each $rt^i_j \in RT_i$, rt^i_j has a label $<N,E_e,E_g,\psi,\beta>$, such that N is a unit, E_e and E_g are environments, ψ a set of places, β a set of arcs.

2.2 Dynamic of labeled reconfigurable nets

Let $LRN = \{E_1, E_2, \ldots, E_p\}$ be a labeled reconfigurable net,

Let $E_i = (GP^i, RP^i, U^i, A^i)$ be an environment in LRN,

- $GP^i = \{gp_1^i, gp_2^i, \ldots, gp_s^i\}$; "*guest places* " in the environment E_i;
- $RP^i = \{rp_1^i, rp_2^i, \ldots, rp_p^i\}$; "*resource places*" in the environment E_i;
- $U^i = \{ N_1^i, N_2^i, \ldots N_k^i\}$; the set of units in the environment E_i;
- $A^i \subseteq GP^i \times starts^i \cup RP^i \times T^i \cup T^i \times RP^i$, where:

  $Sarts^i = \{start^1, start^2, ..., start^k\}$ and $T^i=\{ST^i_1, ST^i_2, ..., ST^i_k\}\cup\{RT^i_1, RT^i_2, ..., RT^i_k\}$

Let RT^i_j be the non empty set of reconfigure transitions associated with the net N^i_j :

$RT^i_j=\{rt^j_1, rt^j_2, \ldots, rt^j_r\}$.

Let $rt^j_{m < N, Ee, Eg, \psi, \beta>}$ be a reconfigure transition in RT^i_j, such that :

- $E_e=(GP^e, RP^e, U^e, A^e)$; the emitter (sender) environment;
- $N=(P, T, A)$ and $N \in U^e$; The unit to be send;
- $E_g=(GP^g, RP^g, U^g, A^g)$; The guest (destination) environment;
- $\psi \subseteq RP^e$; $\psi=\psi_r \cup \psi_c$. (ψ_r denotes removed places and ψ_c denotes cloned places).
- β is a set of arcs. $\beta \subseteq RP^e \times T \cup RP^g \times T$.

Let *strt* be the start transition of N.

- **Conditions to fire $rt^j_{m<N, Ee, Eg, \psi, \beta>}$:**

In addition to the known conditions (conditions to fire a transition in classical Petri nets), we impose that there exists a *free* place p_g in GP^g; witch means: for each $t \in starts^g$, $(p_g,t) \notin A^g$.

- **After firing rt^j_m:**

In addition to the known post-condition of a transition firing in classical Petri nets, we add the following post-condition:

LRN will be structurally changed such that:

If E_e and E_g denote the same environment then LRN will be not changed;

Else:

1) $U^g \leftarrow U^g \cup \{N\}$; $U^e \leftarrow U^e/\{N\}$; the unit N is added to the set of units in the environment E^g;

2) $A^g \leftarrow A^g \cup (p_g, strt)$; the start transition in N will have as input place the place p_g; this must model that the environment E^g receives the unit N and, this last one can start ist execution now.

3) Let $DA = \{(a, b) \in A^e / (a \notin \psi \text{ and } b \notin \psi) \text{ and } ((a \in N \text{ and } b \notin N) \text{ or } (a \notin N \text{ and } b \in N))\}$, $A^e = A^e - DA$. DA –deleted arcs- to be deleted after moving N. These deleted arcs can model the disappear and the disconnection of the unit N from its context in the sender environment;

4) $RP^g \leftarrow RP^g \cup \psi$; $RP^e \leftarrow RP^e/\psi_r$. The places ψ_r can model the migration of a transferable resource between the two environments;

5) if A_{LRN} is the set of arcs in LRN, $A_{LRN} \leftarrow A_{LRN} \cup \beta$. These new arcs can model the rebinding between the two environments. These rebinding are necessary for resources access.

2.3 Examples of Modeling

A mobile code system is composed of execution units (EUs), resources, and computational environments (CEs). EUs will be modeled as units and computational environments as environments. Modeling resources requires using a set of places.

Reconfigure transitions model mobility actions. The key in modeling mobility is to identify the label associated with the *reconfigure transition*. We must identify the unit to be moved, the target computational environment and the types of binding to resources and their locations. This label depends on the kind of mobility.

In general, a reconfigure transition *rt* is always labeled <EU, CE, CE', ψ, β>, such that:

- EU: the execution unit to be moved.
- CE, CE': respectively, resource and target computational environments.
- ψ: will be used to model transferable resources. So ψ is empty if the system has no transferable resource.
- β: models bindings after moving.

The execution unit that contains *rt* and the EU that represents the first argument in the label will be defined according to the three design paradigms: remote REV) evaluation, code on demand (COD), and mobile agent (MA).

2.3.1 Remote Evaluation

In remote evaluation paradigm, an execution unit EU_1 sends another execution unit EU_2 from a computational environment CE_1 to another one CE_2. The reconfigure transition *rt* is contained in the unit modeling EU_1, and EU_2 will be the first argument in *rt*'s label.

Let us consider two computational environments E_1 and E_2. Firstly, E_1 contains two execution units EU_1 and EU_2; E_2 contains an execution unit EU_3. The three execution units execute infinite loops. EU_1 executes actions $\{a_{11}, a_{12}\}$, EU_2 executes actions $\{a_{21}, a_{22}, a_{23}\}$, and EU3 executes actions $\{a_{31}, a_{32}\}$. a_{21} requires a transferable resource TR_1 and a non-transferable resource bound by type PNR_1 which is shared with a_{11}. a_{22} and a_{12} share a transferable resource bound by value VTR1, and a_{23} requires a non-transferable resource NR_1. In E_2, EU_1 requires a non-transferable resource bound by type PNR_2 to execute a_{31}. PNR_2 has the same type of PNR_1.

The system will be modeled as a labeled reconfigurable net LRN. LRN contains two environments E_1, E_2 that model the two computational environments (CE_1 and CE_2). Units EU_1 and EU_2 will model execution units EU_1 and EU_2, respectively. In this case, the unit EU_1 will contain a *reconfigure transition* $rt_{<EU2,E1,E2,\psi,\beta>}$; such that:

1. $E_1 = (RP_1, GP_1, U_1, A_1)$; $RP_1 = \{TR_1, PNR_1, VTR_1, NR_1\}$. $U_1 = \{EU_1, EU_2\}$;
2. $E_2 = (RP_2, GP_2, U_2, A_2)$; $RP_2 = \{ PNR_2\}$. $GP_2 = \{PEU_1\}$.
3. $\psi_r = \{TR_1\}$, $\psi_c = \{VTR_1\}$;
4. $\beta = \{(PEU_1, str_2), (PNR2, a_{21}), (NR_1, a_{23})\}$.

Figure 1 shows the model this system before the migration and after the migration.

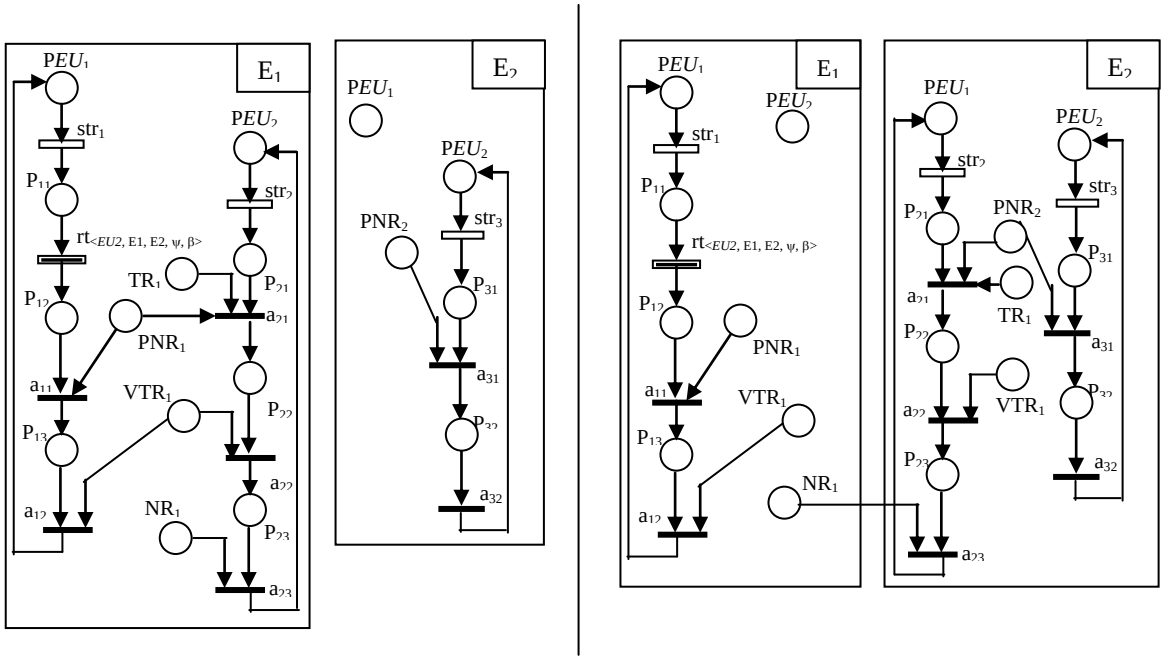


Figure III. 1. REV-Model before and after firing rt

2.3.2 Code On Demand

In code-on-demand paradigm, an execution unit EU_1 fetches another execution unit EU_2. The reconfigure transition *rt* is contained in the unit modeling EU_1, and EU_2 will be the first argument in *rt*'s label. If we reconsider the above example, the unit EU_1 will contain a reconfigure transition $rt_{<EU2, E2, E1, \psi, \beta>}$.

The transition $rt_{<EU2, E2, E1, \psi, \beta>}$ means that EU_1 will demand EU_2 to be moved from E_2 to E_1. In this case, $\psi=\{TR_1, VTR_1\}$, $\beta=\{(PEU_2, str_2), (PNR2, a_{21}), (NR_1, a_{23})\}$.

Figure 2 shows the model proposed to model this system.

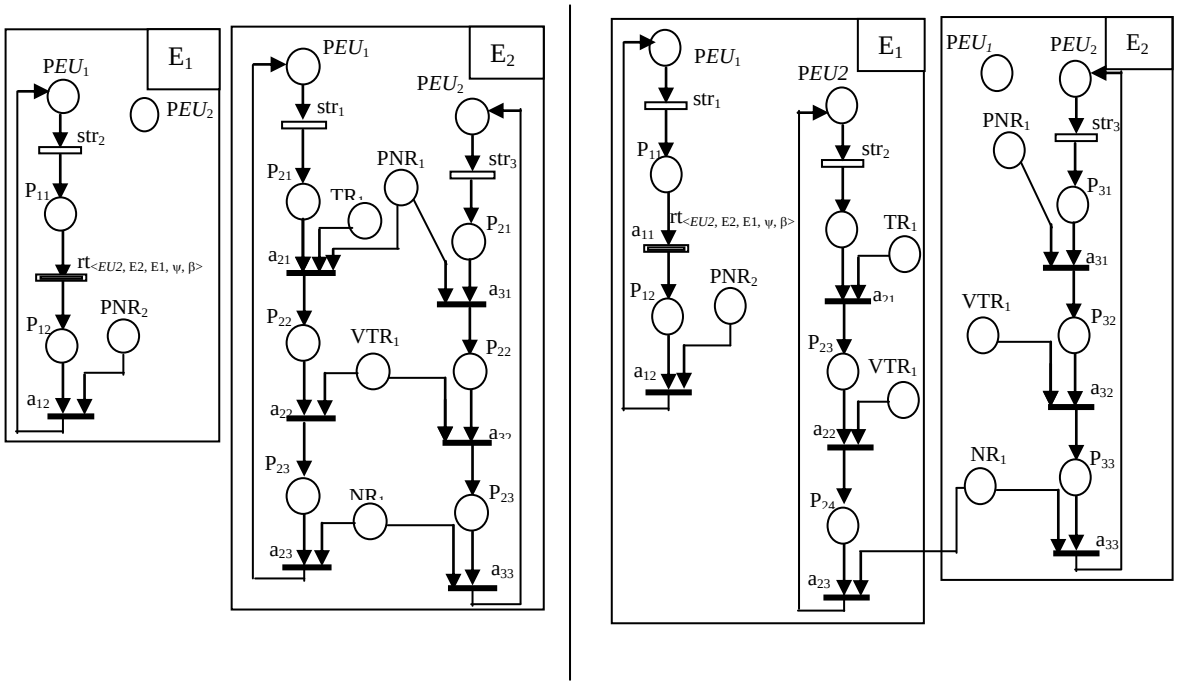


Figure III. 2. COD-Model before and after firing rt

2.3.3 Mobile Agent

In mobile agent paradigm, execution units are autonomous agents. The agent itself triggers mobility. In this case, *rt* –the *reconfigure transition*- is contained in the unit modeling the agent and EU (the first argument) is also this agent.

Let E_1 and E_2 two computational environments. E_1 contains two agents, a mobile agent MA and a static agent SA_1; E_2 contains a unique static agent SA_2. The three agents execute infinite loops. MA executes actions $\{a_{11}, a_{12}, a_{13}\}$, SA_1 executes actions $\{a_{21}, a_{22}, a_{23}\}$, and SA_2 executes actions $\{a_{33}, a_{32}\}$. To be executed, a_{11} require a transferable resource TR1 and a non-transferable resource bound by type PNR_1 which is shared with a_{21}. a_{12} and a_{22} share a transferable resource bound by value, and a_{13} and a_{23} share a non-transferable resource NR_1. In E_1, SA_2 requires a non-transferable resource bound by type PNR_2 to execute a_{32}. PNR_2 has the same type of PNR_1.

The system will be modeled as a labeled reconfigurable net LRN. LRN contains two environments E_1, E_2 that model the two computational environments. In this case the unit *A* that models the mobile agent A will contain a reconfigure transition rt < *A*, E_1, E_2, ψ, β >; such that:

1. E_1 =(RP_1, GP_1, U_1, A_1); RP_1 contains at least four places that model the four resources. Let TR_1, NR_1, PNR_1 and VTR_1 be these places. GP_1 contains at least a free place PA_1 modeling that *A* can be received, and U_1={*A*}.
2. E_2=(RP_2,GP_2, U_2, A_2); RP_2={PNR_2}, GP_2={PA_2}.
3. ψ_r={TR_1}, ψ_c={VTR_1};
4. β={(PA_2, str_1), (PNR2, a_{11}), (NR_1, a_{13})}.

Figure 3 shows the model proposed to model this system.

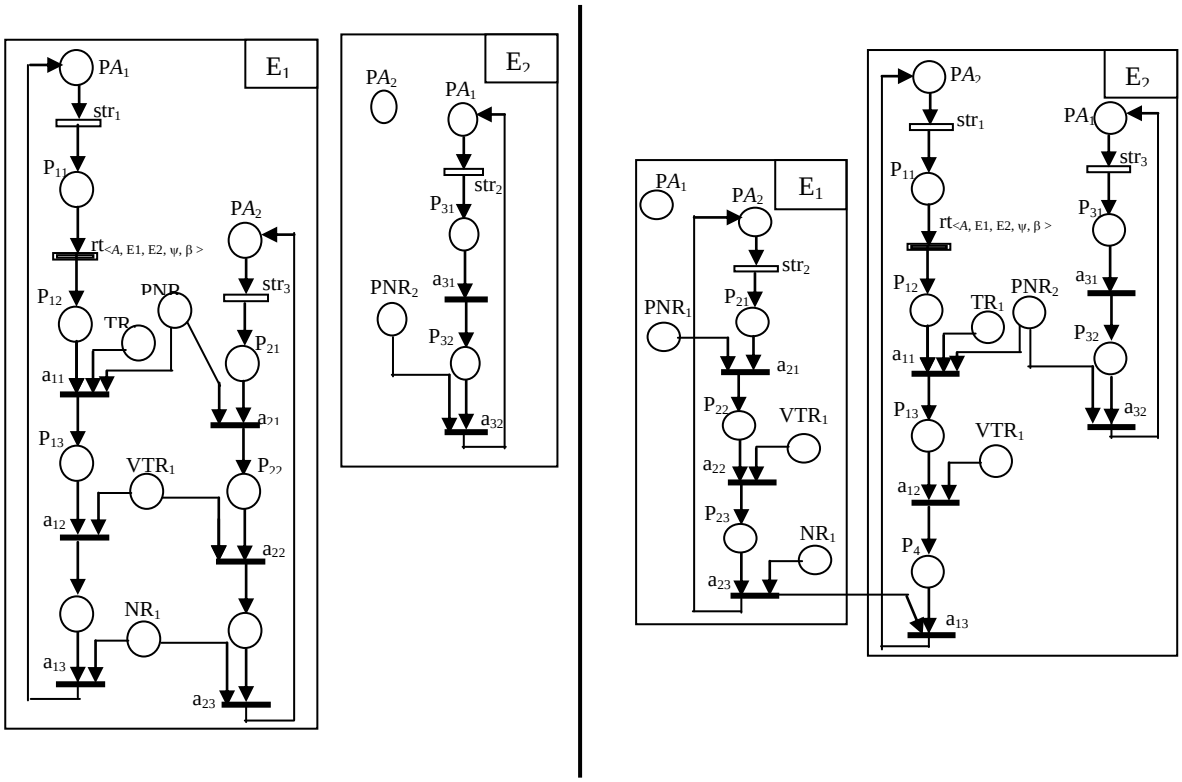


Figure III. 3. MA-Model before and after firing rt

2.4 Shortcomings and Extensions

This first formalism is a simple formalism dedicated to mobile code systems. It offers an intuitive and very explicit way to model migration with all its forms. The bending between environments is modeled through connections (arcs) that can be updated after the migration of some unit. The most shortcoming of this model is due to the idea of the labels associated to the reconfigure transition. The data given as a label are necessary to realize the mobility, however these labels make the model unnatural and far from the intuition of Petri nets. This makes the formalism to lose the background formal which is necessary to do analysis.

One solution is to push the data given in labels to input places of the reconfigure transition. So the environments, the unit to move, the necessary bindings can be seen as tokens to be stored firstly in some input places. This idea imposes to extend the first formalism to another one, where it is possible to define data in places, so to define variables and types. The extension proposed is the *Colored Labeled Reconfigurable Nets* [45].

3 Colored Reconfigurable Nets

Colored reconfigurable nets [45] are an extension of labeled reconfigurable nets. Informally, a colored reconfigurable net is a set of *environments* (blocs of *units*). Connections between these environments and their contents can be modified during runtime. A *unit* is a specific Petri net. A unit can contain three kinds of transitions (a unique *start* transition: , a set of ordinary transitions: , a set of calculi transition: and a set of reconfigure transitions:).

When a *reconfigure transition* is fired, a net N will be (re)moved from an environment E

towards another environment E'. The net N, the environment E and E' are defined by a *calculi transition* witch must always precedes the reconfigure transition. Here after we redefine the concepts of: unit, environment then we give the definition of a colored reconfigurable net.

3.1 Formal Definition

To define colored reconfigurable nets, we redefine firstly the unit and the environment.

Definition 3.1 (Unit) :

A unit is a net U=(Σ, P, T, A, C, E)

Σ : a finite set of types (colors); we denote by *expr* the set of expression that can be written using variables in sets of Σ.

P: a finite set of places;

T: a finite set of transitions. We have $T=\mathcal{T}\cup\mathcal{C}\cup\mathcal{R}$. Where

$\mathcal{T}$: a set of ordinary transitions, $\mathcal{T}=\{t_1, \ldots, t_n\}$. This set must contain a unique transition that we call a *start* transition. We denote this transition as *strt*,

$\mathcal{C}$: a set of calculi transitions, $\mathcal{C}=\{c_1, \ldots, c_m\}$,

$\mathcal{R}$: a set of reconfigure transitions, $\mathcal{R}=\{r_1, \ldots, r_p\}$.

A: a set of arcs

C: a color mapping from P to Σ. C joins to each place *p* a color *c* that we note C(*p*).

E: an expression mapping from A to *expr*.

Definition 3.2 (Environment): an environment E is a quadruplet E=(GP, RP, U, A)

- $GP = \{gp_1, gp_2, \ldots, gp_s\}$ a finite set of specific places : "*guest places* ";
- $RP = \{rp_1, rp_2, \ldots, rp_s\}$ a finite set of specific places : "*resource places*";
- $U = \{ N_1, N_2, \ldots N_k\}$ a set of nets, where $T_1, T_2, \ldots, T_k$ are the sets of their transitions and $StrT=\{strt^1, strt^2, \ldots, strt^k\}$ is the set of their start transitions.
- A : a set of arcs, $A \subseteq GP \times StrT \cup RP \times T$. Such that: $T=T_1\cup T_2 \cup \ldots \cup T_k$

Remark : we say that a unit U is in an environment E iff the net U is a subnet of the net E.

Definition 3 (Colored reconfigurable nets):

A colored reconfigurable nets (CRN) is couple N=(E, A), such that:

E: a finite set of environments;

A: a finite set (probably empty) of arcs; these arcs connect places (resp. transitions) from one environment to other transitions (resp. places) in another environment.

3.2 Dynamic of colored reconfigurable nets

To introduce the dynamic of CRN we consider three types (colors): P(set of places), N(set of nets), and B(set of arcs). We denote respectively by P^*, N^*, B^* the three multi-sets of types P, N, B. We focus on the semantic of the *calculi* and the *reconfigure* transition.

Semantics of calculi transition:

A calculi transition must take as input three tokens of type N(two environments and one unit, the unit must be in one and only one of the two environments). Firing the calculi transition provides a token in the multi-sets $< N^*, P^*, B^* >$. We can say that a calculi transition uses a set of nets to computes some arcs and places. At the output, it provides a composite token of the input nets and the computed arcs and places. In general, this token is used by a reconfigure transition.

If *t* is a *calculi transition*, and E_1, E_2 (represent two environments), U (represents a unit, U is in E_1) are the input nets, once t is fired it produces a token $<U+E_1+E_2, P, A>$ such that *P* and *A* are two multi-sets that can be defined like this:

$P=\{p \in P_{E1} / \ p \notin P_U$ and $\exists\ t \in T_U$ such that $(p,t) \in A_{E1}$ or $(t,p) \in A_{E1}\}$,

and

$A=\{a \in A_{E1} / \ a \notin A_{E1}$ and $\exists\ (t,p)\ T_{E1} x P_U \cup T_U x P_{E1}\}$.

Where P_N, A_N and T_N denote respectively places, arcs and transitions of a net N.

Informally, *P* can model places that contain resources in the sender environment E_1. These resources are used by the unit U before its migration. The mutli-set *A* models connections between the unit U and the environment E_1.

Semantics of reconfigure transition:

The objective of a reconfigure transition is to reconfigure the structure of the net. To be fired, a reconfigure transition takes as input a token in the multi-sets : $< N^*, N, P^*, B^* >$. Firing a reconfigure transition will update the structure of the colored reconfigurable nets that contains this transition in the following semantics:

If *rt* is a reconfigure transition and $<U+E_1, E_2, P, A>$ is an input token, to fire *rt* we impose that there exists a *free* place p_g in GP_{E2}; witch means:

$$\text{For each } t \in \text{str}T_{E2}\ (p_g,t) \notin A_{E2}.$$

Where $\text{str}T_{E2}$ denotes the set of the start transitions of all units defined in the environment E_2.

Once this condition is satisfied, firing *rt* changes N structurally such that:

- If E_1 and E_2 denote the same environment then N will not be changed;

  Else:

  1) The net U is removed from the net E_1: $U_{E2} \leftarrow U_{E2} \cup \{U\}$;

2) The net U is added to the environment E_2: $U_{E1} \leftarrow U_{E1}/\{U\}$;

3) $A_{E2} \leftarrow A_{E2} \cup (p_g, strt)$; such that *strt* is the start transition for U.

4) Some elements of *P* are transformed from E_1 towards E_2, some others are cloned and some others will not be changed (resp for elements in *A*). These elements depend on the modeling case.

3.3 Examples of Modeling

As an example, we will redo the modeling of the example of Mobile agent system, presented above with LRN. The modeling of the two other examples can be done in the same manner.

We use the same description given above for a mobile agent system. The system will be modeled as a colored reconfigurable net N. N contains two environments E_1, E_2 that model the two computational environments (CE_1 and CE_2). Units A_1 A_2 and A_3 will model MA, SA_1 and SA_2, respectively. In this case, the unit A_1 will contain a *reconfigure transition rt* and a calculi transition *cu*.

1. $E_1=(RP_1, GP_1, U_1, A_1)$; $RP_1=\{TR_1, PNR_1, VTR_1, NR_1\}$. $U_1=\{EU_1, EU_2\}$;
2. $E_2=(RP_2, GP_2, U_2, A_2)$; $RP_2=\{ PNR_2\}$. $GP_2=\{PEU_1\}$.
3. *ct* will take as input tokens : E_1, A_1 and E_2. *ct* will provide the token : $<A_1+E_1, E_2, P, A>$. such that

   $P=TR_1+VTR_1$

   $A=(NR_1,a_{23})+(PNR_2,a_{21})$
4. *rt* takes as input $<A_1+E_1, E_2, P, A>$ and will remove A_1 and places in *P* from E_1 towards E_2. Arcs in *A* will be added to the N.

In the figure III.4, the types of places P^1, P^2, P^3 is N(set of nets). P^1 contains A_1, P^2 contains E_2 and P^3 contains E_2. The type of place P_{11} is <N*, N, P*, B*>. Figure III.4 shows the model of this system before and after the migration of a mobile agent.

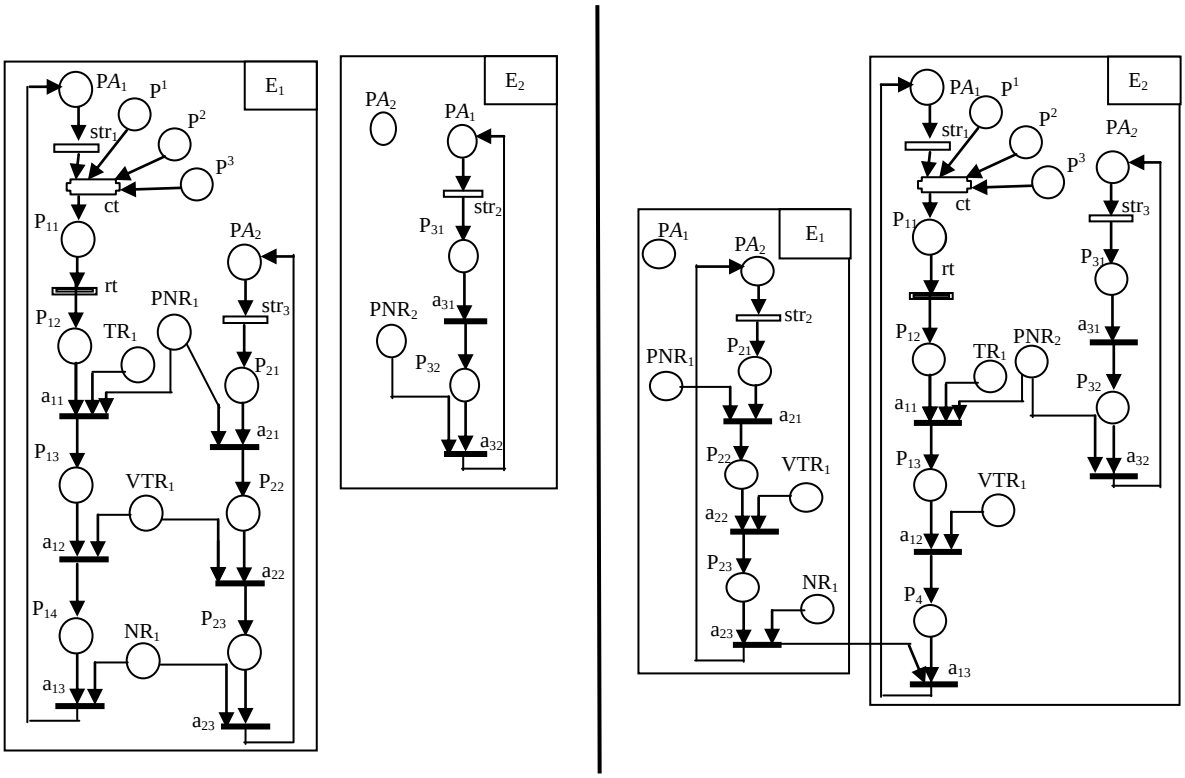


Figure III. 4. MA-Model (modeled with CRN) before and after firing rt

3.4 Shortcomings and Extensions

The extra data used as labels in the LRN model is resolved by the use of types in the CRN models. The types offer the possibility of interpreting mobility as the migration of some parts of a net from one environment (which is also a net) toward another. The net to be moved, the send and the destination are all expressed as complex data defined as tokens in some places. The migration requires firstly defining the nets to be moved, and the changes that will be applied to the two environments. This step is realized firstly by one transition (the calculus transition), then the reconfiguration is insured by a second transition (the reconfigure one).

The problem now is the interpretation of the effect of the two transitions. The transition which does a calculus then provides a set of data, and the transition that achieve the reconfiguration of the net. A formal interpretation of the two effects into the Petri nets known operation is required to insure the formal verification of these models using the formal methods applied early in Petri nets. In the absence of this formal interpretation, all that one can do is to realize a simulation of the model or to compute a reachablity tree and does the possible verification. The computation of the reachability tree must consider the specific transitions. Each time, when we have a *calculus transition* or a *reconfigure transition*, one must call to specific procedures that have a different effect than ordinary transitions. These procedures will calculate data using the semantics associated to the calculus transitions or will change the structure of the net using the semantics of the reconfigure transition.

It is clear that a reachability tree is this case will not be like ordinary reachability tree. each node in our reachability tree must be composed by to component :a marking vector that denotes the marking of each place, and a structure state that represents the structure of the net

at the current state. The state of a reconfigurable system (like a mobile code system) is no more modeled by the marking of the net, but also by the structure of the net. The marking and the structure are the two dynamic and the two change over time.

The simulation of the models and the computation of a reachabililty tree seem to be two techniques that can be applied to do analysis for models in LRN or CRN. But really a formal interpretation needs to be defined. A formal interpretation of the extra-natural transitions seems to be required to insure that the proposed model is always a Petri net or can be translated to a Petri net. To insure this, we will proceed as follows:

- Firstly, we think that the above proposed models are focused on mobility which represents a specific case of a large class of systems, which are reconfigurable systems. A reconfigurable system can be as a logic system (code mobility) or a physic system (mobile robots, mobile wireless networks, …). The mobility is seen as a specific reconfigurablity. It seems to be more adequate to propose formalism for reconfigurable systems. The reconfigurability of a system will not, necessary, require the migration of a whole unit form one environment to another one. The reconfigurability can be so simple that it consists of a small modification of the structure of some unit in a system (logic or physic). These kinds of modifications motivate us to think of some formalism where the reconfiguration is defined into a low level. Considering a net as a graph, a low level reconfiguration lets the net to modify its structure by adding or deleting one component (a place, a transition, or an arc). This small modification can be used after to realize more significant reconfiguration, which is the case in mobile code systems. So, the proposition can be seen as a reconfigurable net with a high granularity. We will present this proposition in the next section (section four), as Flexible nets.

- Considering the formal interpretation of the reconfiguration, we will propose to prove that all *reconfiguration behaviors* (implying the modification of the structure of the net during its execution) can be encoded into classical Petri nets behaviors. This result will be more explained in the chapter four.

4 Flexible Nets: The mature idea

Flexible Nets (FN) is an extension of colored Petri Nets [47, 49]. In FN, the places, the transitions and the arcs are objects that can figure as marking of places. These objects can be signed: positive or negative. We introduce three sorts: P (for place), T (for transition), and A (for arc). These three sorts can contain negative as positive objects. We abandon the use of reconfigure transitions. All transitions can change the structure of the net depending on the expressions labeling input arcs of these transitions. By using signed objects, we offer the possibility for adding (or deleting) nodes to (from) the net, when a transition is fired. The reconfiguration of the net by adding or deleting nodes will be interpreted as an internal operation defined in the sorts P, T and A. In the next paragraphs, we will present this internal operation and how it reconfigures a net. The initial marking of an added place, the guard associated to an added transition, and the expression labeling an added arc must be present when the transition that will add these nodes is fired. These three pieces of information can be modeled as data presented in the input places of a transition. Finally, we consider that transitions can be temporized.

4.1 Formal Definition

A Flexible Net N is a 9-tuple (Σ, P, T, A, C, G, E, I), where:

- Σ: a set of types (Colors). We denote by Σ* the set of all muti-sets of the set Σ;
- P: a set of places;
- T: a set of transitions;
- A: a set of arcs. A⊆(TxP) ∪(PxT);
- C: a color function associated to each place. C: P→Σ. For each place p, C associates a unique color C(p);
- G: a guard function associated to each transition. G: T→Exp. Where Exp is the set of all Boolean expressions that can be constructed using constants and variables defined in types Σ;
- E: an expression function that associates to each arc *a* in A an expression E(*a*);
- I: is an initial state of the net. I=<M_0, S_0>, where M_0 is the initial marking of places P. M_0: P→Σ*. S_0 is the initial structure of the net. We take S_0=P∪T∪A.

In section 4.2 and in section 4.3 of this first part of the chapter, variables are always written as *italic* letters and constants as regular letters. Expressions are always written between angle brackets. The expression <> denotes the empty expression. The sets P, T and A are considered as types in Σ. We use the symbols P, T and A, to denote the names of these types and also to denote the sets that contain the elements (places, transitions and arcs) in the net N. This means that these sets can change over the execution of the net, but in some occurrences P, T, and A refer to the types which are abstract notions. These types contain names of places, transitions and arcs. These names can be signed. For example, we note p as an unsigned place, and +p, and -p as a signed one. We define the internal operation ⊕ in sets P, T, and A. We denote by ∅ the neutral element for the operation: ⊕. We adopt the following proprieties for this operation:

Let e, e', e'' be three signed elements defined strictly in one of the three above sets: P, T, or A, we have:

- e⊕e'= e'⊕e;
- (e⊕e')⊕e''= e'⊕(e⊕e'');
- e⊕-e=-e⊕e=∅;
- e⊕∅=e;

If E={e_1, e_2, e_3} is a subset of one of the three sets P, T, or A. We allow that E can also be written as an expression E_{term}=e_1⊕e_2⊕e_3. In this semantics, the expression e⊕e' denotes the set {e, e'}, and the expression e⊕-e denotes the set {e}/{e'} which is briefly the set {e}. The neutral element ∅ denotes the empty set {}.

In the set of types Σ={C_1, C_2, …}, we can have complex types. If *C* is a complex type in Σ, then every element *e* in *C* can be written as a tuple <e_1, e_2, e_3, …, e_n>, such as *n* is the arity of *e*. In this case, we denote by e_i the sub-element of order *i* in the element *e*, and we denote by *Type*(e_i) the type of this sub-element. In a tuple <e_1, e_2, e_3, …, e_n> which appears in a marking of a place p, we suppose that:

- if *Type*(e_i)=P (which means that e_i is a place) and e_i is positive , then e_{i+1} and e_{i+2} must be the two expressions that represent respectively, the initial marking of e_i and the type of e_i. If e_i is negative e_{i+1} and e_{i+2} will be empty expressions;
- if *Type*(e_i)=T (which means that x_i is a transition) and e_i is positive, then e_{i+1} must be the expression that represents the guard of e_i. If e_i is negative e_{i+1} can be an empty expression;
- if *Type*(e_i)=A (which means that e_i is an arc) and e_i is positive, then e_{i+1} must be the expression labeling the arc e_i. If e_i is negative e_{i+1} will be an empty expression;

4.2 Firing Rules

Let N be a Flexible Net, and *t* a transition in T. As in CPN [31], we denote by °*t* the set of input places of the transition *t*, and by *t*° the set of output places of the transition *t*. Let I_0=<M_0, S_0> be the current state of N. Firing *t* changes I_0 towards I_1=<M_1, S_1>. We denote this as : <M_0, S_0>$\rightarrow^{t}$<M_1, S_1>.

Preconditions to fire *t.* *t* can be fired iff there is a unification υ such that $M_0(p) \geq E(p, t)[\upsilon]$, for each *p* in °*t*, and G(*t*) is true.

Post-conditions of firing *t.* after the firing of *t*, N will transit from its current state I_0 to another state I_1=<M_1, S_1>. For each *p* in °*t*, we will have: $M_1(p)=M_0(p)-E(p,t)[\upsilon]$. For each *p* in *t*°, we will have: $M_1(p)=M_0(p)+E(t,p)[\upsilon]$. S_0 (which is $P_0 \cup T_0 \cup A_0$) will be updated to $S_1=P_1 \cup T_1 \cup A_1$, with their terms $P1_{term}$, $T1_{term}$, $A1_{term}$.

For each *p* in °*t*, if E(*p*, *t*) is the tuple <e_1, e_2, …, e_n>, then for each e_i in e_1, e_2, …, e_n:

- if *Type*(e_i)=P then $P_{1term}= P_{0term} \oplus e_i[\upsilon]$, $C(e_i[\upsilon])=e_{i+1}[\upsilon]$, and $M_1(e_i[\upsilon])=e_{i+2}[\upsilon]$;
- if *Type*(e_i)=T then $T1_{term} = T0_{term} \oplus e_i[\upsilon]$ and $G(e_i[\upsilon])=e_{i+1}[\upsilon]$;
- if *Type*(e_i)=A then $A1_{term} = A0_{term} \oplus e_i[\upsilon]$ and $E(e_i[\upsilon])=e_{i+1}[\upsilon]$.

The following three examples are presented to clarify the semantics of this formalism. We start by the example of Figure III.5.

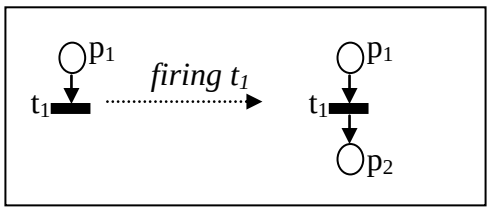


Figure III. 5. Flexible Nets first example

In Figure III.5, firing the transition t_1 will change the structure of the net. In the new structure, a new place p_2 and a new arc (t_1, p_2) are added to the original structure. To ensure this behavior, we must have at the initial state of the net:

- $M_0(p_1)$=<+p_2, <>, "integer", +(t_1,p_2), <1>>. This marking will allow the creation of a new place p_2 which type is integer with an empty initial marking and a new arc (t_1,p_2) which is labeled <1>.
- (p_1,t_1) is labeled <p_1, *mark_*p_1, *Type*, a_1, *exp_*a_1>. Where: p_2, is a variable of type P, a_1, is a variable of type A, *mark_*p_2, *Type*, *exp_*a_1 are variables of type expression.

From this initial state, it is clear that t_1 can be fired with respect to the substitution: υ=[p_1←+p_2, *mark_*p_1←<>, *Type*←"integer", a_1←+(t_1,p_2), *exp_*a_1←<1>]

System in Figure III.6 has a more complex behavior.

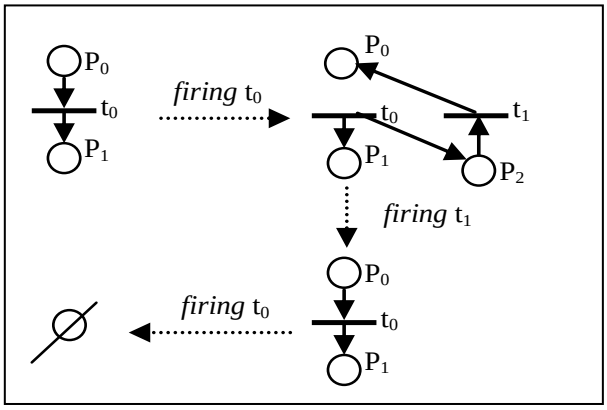


Figure III. 6. Flexible Nets second example.

In this case, the system starts by firing t_0 which will delete the arc (p_0, t_0) and will add: the place p_2, the transition t_0, and the two arcs {(t_0, p_2), (p_2, t_1)}; then, from this state, t_1 can be fired which restitutes the first structure (by deleting the new added nodes and restituting the arc (p_0, t_0)), and finally the t_0 destroys the net. To facilitate this presentation, types of places will not appear in the following expressions. Initially, we must have:

- $M_0(p_0)$=<-(p_0,t_0), p_2, <-t_1, -p_2, (p_0,t_0),
  <*a_1, p_1, mark_p_1, t_1, grd_t_1, p_2, mark_p_2, a_2, exp_a_2, a_3, exp_a_3, a_4, exp_a_4*>
  >,
  t_1, <>, ∅, <>, (t_0, p_2), <∅, ∅, ∅, <>>, (p_2,t_1), <*t_1, p_1, a_1, exp_a_1*>, (t_1, p_0),
  <∅, -p_0, <>, -t_0, <>, -p_1, <>, ∅, <>, ∅, <>, ∅, <>>
  >;
- E(p_0, t_0)=<*a_1, p_1, mark_p_1, t_1, grd_t_1, p_2, mark_p_2, a_2, exp_a_2, a_3, exp_a_3*>;
- E(t_0, p_1)=<1> and $M_0(p_1)$=0.

From this initial state, the transition t_0 is enabled with the substitution:
υ=[*a_1*←-(p_0,t_0), *p_1*←p_2, *mark_*p_1←<-t_1, -p_2, (p_0,t_0), <*a_1, p_1, mark_p_1, t_1, grd_t_1, p_2, mark_p_2, a_2, exp_a_2, a_3, exp_a_3, a_4, exp_a_4*>>,
t_1←t_1, *grd_t_1*←<>, *p_2*←∅, *mark_p_2*←<>, *a_2*←(t_0, p_2), *exp_a_2*←<∅, ∅, ∅, ∅, <>>,
a_3←(p_2,t_1), *exp_a_3*←< *t_1, grd_t_1, p_1, mark_p_1, a_1, exp_a_1*>].

Once t_0 is fired:

- (p_0, t_0) will be deleted,
- p_2 will be added with $M_1(p_2)$=
  <-t_1, <>, -p_2, <>, (p_0,t_0),

$<a_1, p_1, mark\_p_1, t_1, grd\_t_1, p_2, mark\_p_2, a_2, exp\_a_2, a_3, exp\_a_3, a_4, exp\_a_4>$

>,

- t_1 will be added,
- (t_0, p_2) labeled <∅, ∅, ∅, ∅, <>> will be added,
- (p_2, t_1) labeled $<t_1, grd\_t_1, p_1, mark\_p_1, a_1, exp\_a_1>$ will be added,
- (t_1, p_0) labeled <∅, $-p_0$, <>, $-t_0$, <>, $-p_1$, <>, ∅, <>, ∅, <>> will be added,
- $M_1(p_1)$=<1>.

From this second state, t_1 is enabled with the substitution:

υ'=[

$t_1 \leftarrow -t_1$, $grd\_t_1 \leftarrow <>$, $p_1 \leftarrow -p_2$, $mark\_p_1 \leftarrow <>$, $a_1 \leftarrow (p_0, t_0)$,

$exp\_a_1 \leftarrow <a_1, p_1, mark\_p_1, t_1, grd\_t_1, p_2, mark\_p_2, a_2, exp\_a_2, a_3, exp\_a_3, a_4, exp\_a_4>$

].

When t_1 is fired, t_1 and p_2 will be deleted, (p_0, t_0) is restituted, and $M_2(p_0)$=< ∅, $-p_0$, <>, $-t_0$, <>, $-p_1$, <>, ∅, <>, ∅, <>>

From this third state, t_0 is enabled with the substitution:

υ''= [

$a_1 \leftarrow ∅$, $p_1 \leftarrow -p_0$, $mark\_p_1 \leftarrow <>$, $t_1 \leftarrow -t_0$, $grd\_t_1 \leftarrow <>$, $p_2 \leftarrow -p_1$, $mark\_p_2 \leftarrow <>$,
$a_2 \leftarrow ∅$, $exp\_a_2 \leftarrow <>$, $a_3 \leftarrow ∅$, $exp\_a_3 \leftarrow <>$

].

When t_0 is fired, the net will be destroyed.

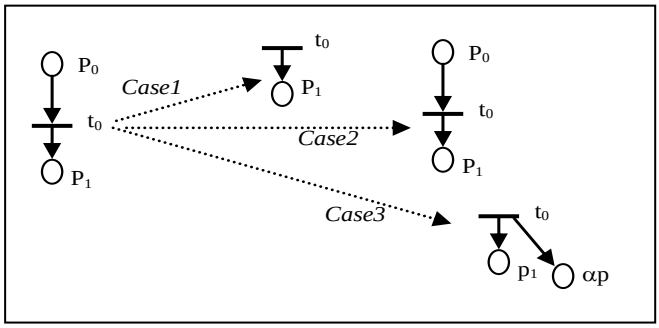


Figure III. 7. Flexible Nets third example.

The example of Figure III.7 treats a particular case. In the current version of FN, we adopt that when a transition *t* tries to delete a node *n* from the structure S, and if *n* does not exist in S, the transition will be fired and the structure will not be changed. But when *t* tries to add a node *n* to the structure S which also contains a node with the same name *n*, we can adopt some conversion (an α-conversion) of the occurrences of *n* to avoid confusion in the generated structure S'.

In the example of Figure III.7, we show the application of an α-conversion when the node to be added existed in the current net. Suppose that the initial marking of p_0 is $<p_0, <>, (t_0, p_0), <1>, -p_0, <>>$. And suppose that $E(p_0, t_0)=<p_1, mark\_p_1, a_1, exp\_a_1, p_2, mark\_p_2>$. In this

initial state, t_0 is enabled with the substitution $\upsilon=[p_1 \leftarrow p_0, exp_1 \leftarrow <>, a_1 \leftarrow (t_0, p_0), exp_2 \leftarrow <1>, p_2 \leftarrow -p_0, mark\_p_2 \leftarrow <>]$. We have three cases: (i) without α-conversion, the system will add a an arc (t_0, p_0) to the net, then it will delete the place p_0, (ii) with an α-conversion that convert all occurrences of p_0, we will have $\upsilon_\alpha=[p_1 \leftarrow \alpha p_0, mark\_p_1 \leftarrow <>, a_1 \leftarrow (t_0, \alpha p_0), exp\_a_1 \leftarrow <1>, p_2 \leftarrow -\alpha p_0, mark\_p_2 \leftarrow <>]$ and the system will add a place then delete this place; so the structure will not change, (iii) with an α-conversion on positive occurrences, we will have $\upsilon_{\alpha+}=[p_1 \leftarrow \alpha p_0, mark\_p_1 \leftarrow <>, a_1 \leftarrow (t_0, \alpha p_0), exp\_a_1 \leftarrow <1>, p_2 \leftarrow -p_0, mark\_p_2 \leftarrow <>]$. In this case, the system will add the place αp_0 and delete p_0.

Now, what kinds of conversion we will adopt? This can be considered as a specialization of the Flexible Nets formalism. We consider that the formalism is generic and such specialization can be chosen by the user depending on the application. This problem can also be avoided if the names chosen are completely different from the current names in the model.

4.3 Examples of Modeling

4.3.1 Example of a Dynamic Join Calculus model

Let review the specification written in the DJC [13] and presented in the chapter two:

ready(printer)|job(file) ▷ printer (file)

ready(laser)| job(1), job(2)

In this specification, once we have a job to be printed and a ready printer we send this job the ready printer. To specify this simple system in the FN formalism, we define two places job, and ready. These two places represented the two terms job and ready in the DJC specification. These two places will be initially marked <1>+<2>, and <laser>, respectively. In the FN formalism (Figure III.8) we will have a new place created to represent every detected ready printer. In the example specified here, we have one ready printer which is laser. So the firing of the transition will create a new place with the name laser, and which will be marked ever <1> or <2>. The name of this created place is a parameter given in the predefined place ready. To specify this behavior, we start by add a transition that create the data which represents the node to be added to the graph at runtime. The transition *t'* in the Figure III.8, create the token <+*printer*, <*file*>>, this one is stored in the place p_1, and will be used by the transition *t* to create a new place with the name stored in the variable printer and with the initial marking<file>.

The idea is that each rule in the DJC which produces new terms (processes) in the system is modeled as two transitions, a first one which will define the nodes to be added, and a second transition which will change effectively the structure of the net, to model the reconfiguration in the DJC specification.

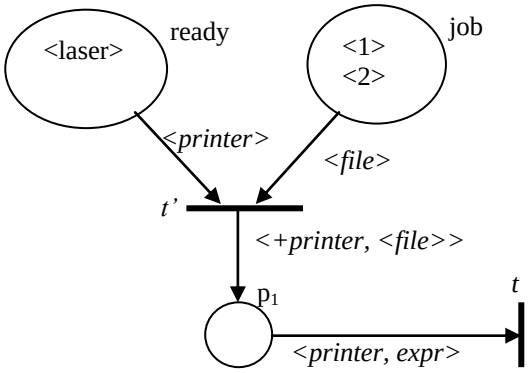


Figure III. 8. The initial configuration.

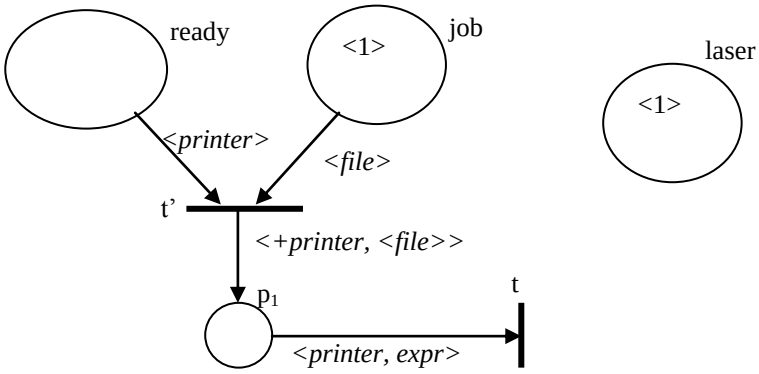


Figure III. 9. The configuration after firing the sequence t',t.

4.3.2 Example of a Mobile Petri Nets model

We take the example presented in the chapter two when we have presented Mobile Petri Nets [20].

Let consider the Mobile net $N = (\upsilon Y)(m, T)$ where:

$Y = \{p_1, p_2, p_3, p_4, p_5\}$, $T = \{t_1, t_2\}$,

$t_1 = \{p_1(x, a, p), p_2(y, z)\} \triangleright \{p(x, y)\}$, $t_2 = \{p_2(p, p')\} \triangleright \{p(1, 2), p'(a, b), p_3(6)\}$

Suppose that the initial marking is $m = \{p_1(1, a, p_5), p_2(p_4, p_5)\}$.

In this example, the firing of the transition t_1 will create the place p which is given as a token in the place p_1. The initial marking of p is <x,y>, where x is given in the place p_1 and y in the place p_2 respectively. In the FN formalism, we have proposed that initial marking of an added place p is defined with this place in the same input place for the transition which will add p.

To model this example using Flexible nets, we will have the initial configuration as showed on Figure III.10. In the specification of Figure III.10, the behavior of the transition t_1 of the Mobile net is reproduced by the sequence "t_0, t_1", and the behavior of the transition t_2 of the Mobile net is reproduced by the sequence "t_3, t_2".

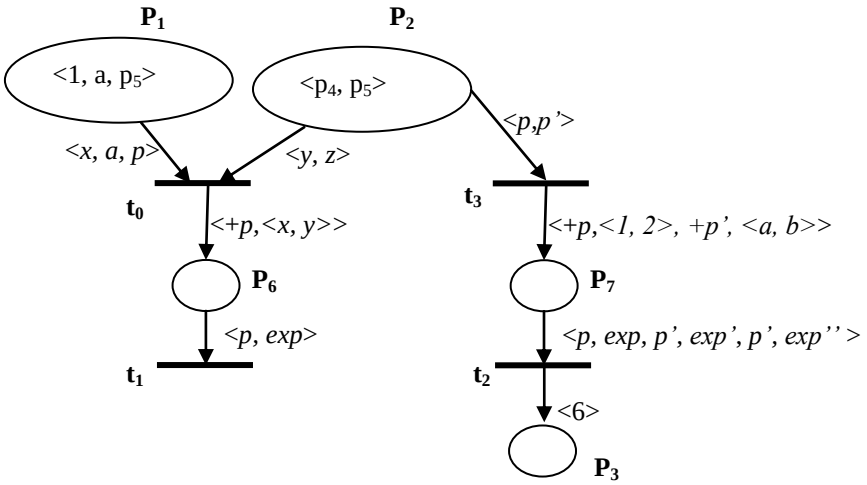


Figure III. 10. Flexible Nets: example of Mobile nets

The behavior of the two transitions t_1 and t_2 in the mobile net is reproduced by the two transition t_1 and t_2 respectively in the Flexible net. The firing of t_1 requires the firing of t_0, and the firing of t_2 requires the firing of t_3.

The firing of the sequence "t_1, t_2" leads to the configuration showed on Figure III.11, and the firing of the sequence "t_3, t_2" leads to the configuration showed on the Figure III.12.

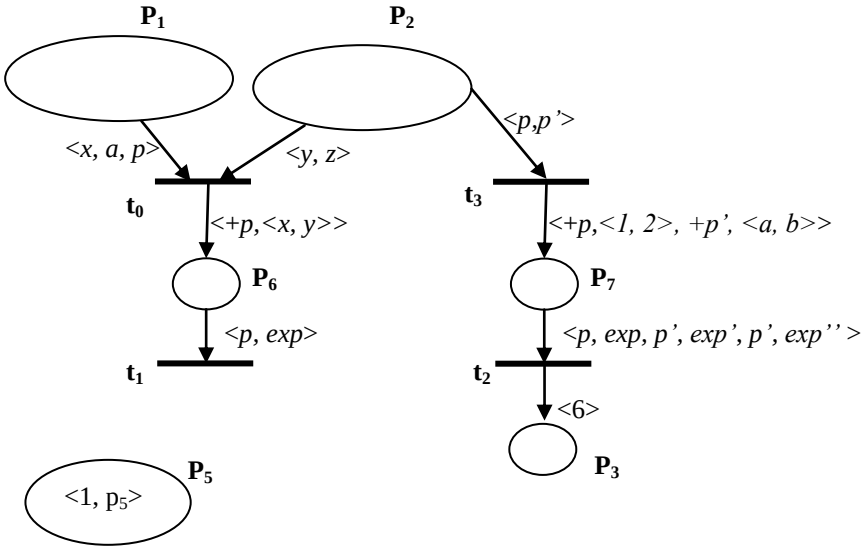


Figure III. 11. Configuration after firing the sequence $t_0 t_1$

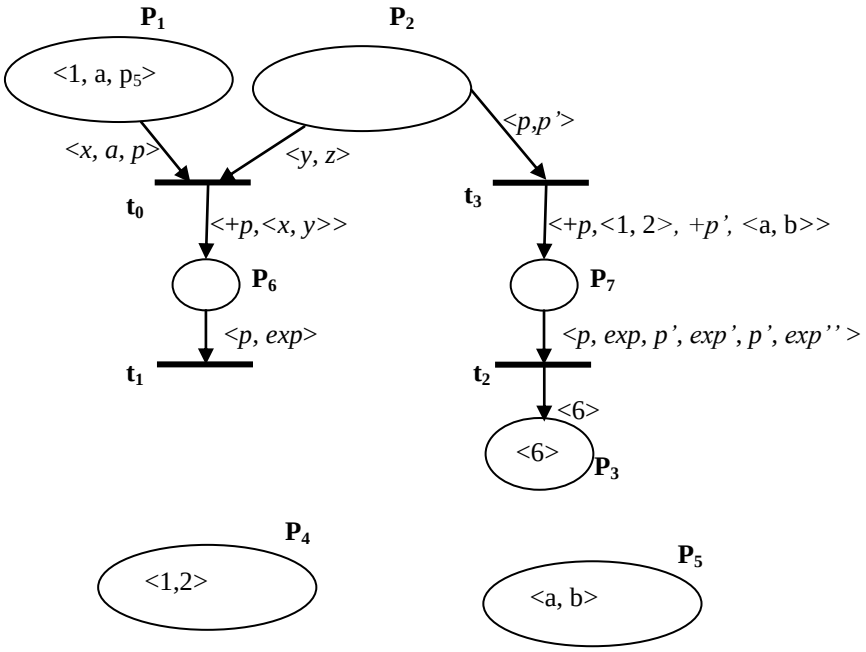


Figure III. 12. Configuration after firing the sequence t_3,t_2

4.3.3 Example of a Dynamic Petri Nets model

Let review the example presented in the chapter two, when we have presented the dynamic net (DN) [20].

Consider the following dynamic net:

$$
\begin{aligned}
&(\upsilon\{A,B\})(\\
&\quad \{A(X) \triangleright\\
&\qquad (\upsilon\{Y\})(\\
&\qquad\quad \{X(W),Y(Z)\} \triangleright W(Z),\\
&\qquad\quad \{A(Y),Y(B)\}\\
&\qquad),\\
&\quad \{A(A),B(B)\}\\
&\quad)
\end{aligned}
$$

The initial marking is *{A(A), B(B)}*.

Like we have explained in the chapter two, when the external transition is fired, it adds a new transition and a new place.

The FN of this example is presented in Figure III.13. In Figure III.13, the behavior of the external transition defined in the DN is reproduced by the sequence "t_0,t_1". The behavior of the internal transition is reproduced by the sequence " t'', t' ". This last sequence (and so its behavior) is defined also in the marking created when the transition t_0 is fired.

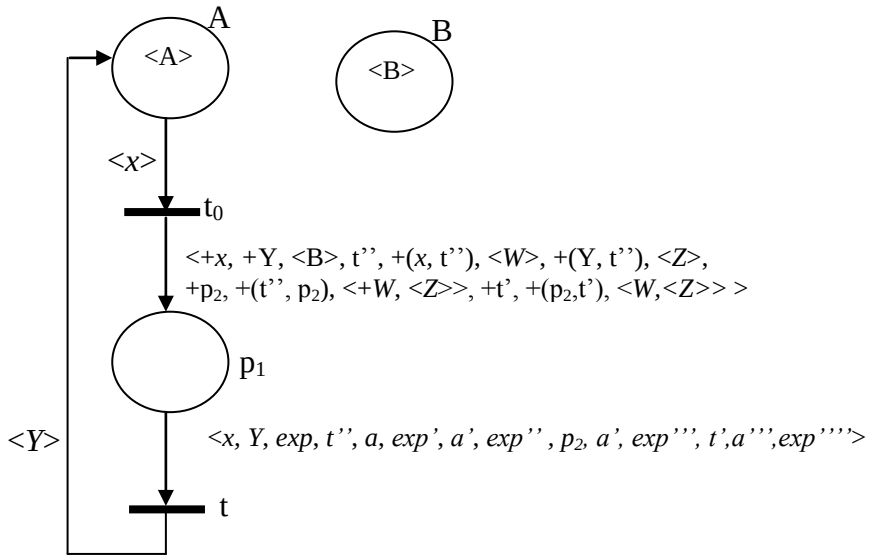


Figure III. 13. The initial configuration

In Figure III.14 and Figure III.15, exp_1=<+x, +Y, <B>, t'', +(x, t''), <W>, +(Y, t''), <Z>, +p_2, +(t'', p_2), <+W, <Z>>, +t', +(p_2,t'), <W,<Z>> > , and exp_2= <x, Y, exp, t'', a, exp', a', exp'' , p_2, a', exp''', t',a''',exp''''>

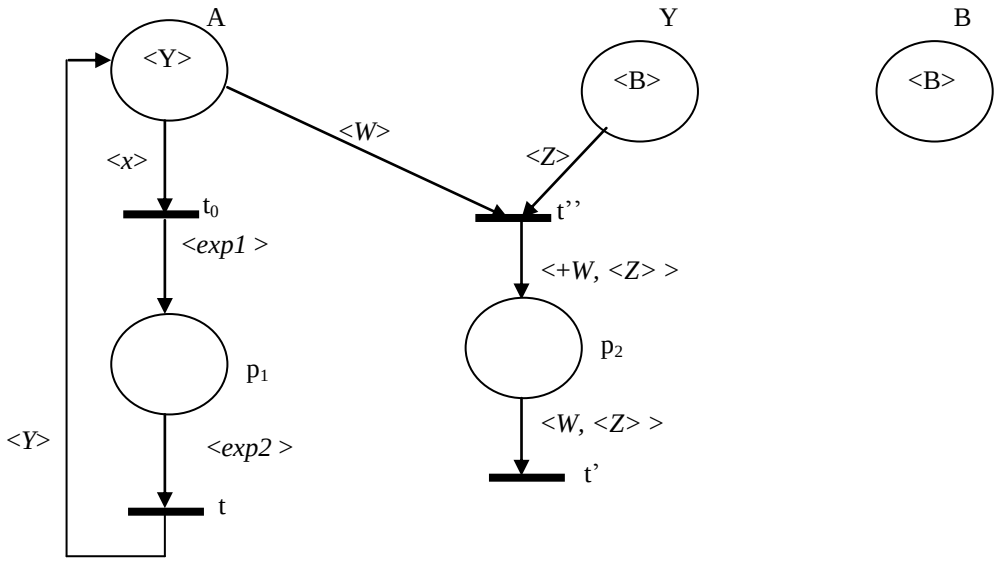


Figure III. 14. The configuration after the firing of the sequence t_0, t

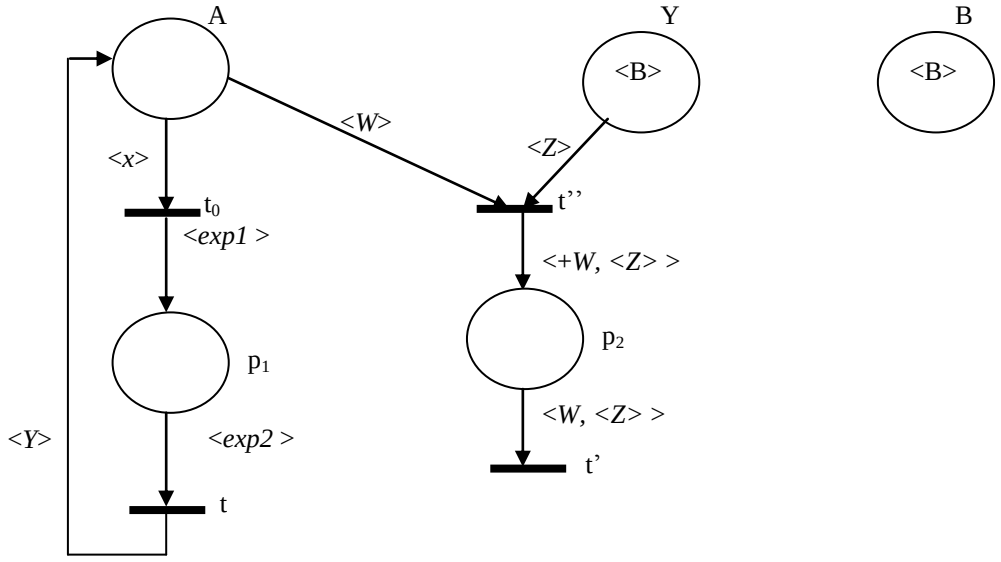


Figure III. 15. The configuration after the firing of the sequence t'', t'

5 Analysis Issues

The use of formal methods finds its motivation in the analysis and verification techniques that can be used with these methods. The proprieties that can be verified depend on the kind of the method used and the modeled system. In case of Petri Nets, some known proprieties are defined: boundedness, safety, reachability …etc. The most of proposed extensions of Petri Nets have some automatic tools to achieve the verification of some proprieties. In the current work, we consider that the same proprieties defined for classical Petri Nets can be studied and extended for Flexible Nets. To analyze these proprieties, we proposed two ways in the current time:

- The first way is based on an extended reachability tree that can be generated automatically. In classical Petri Nets, the reachability tree has as root the initial marking, and as nodes all the reachable marking. Arcs between nodes (marking) are labeled with the transition that transforms some marking (first node) to another marking (second node). In the FN, nodes of this reachability tree are not simple marking but they are states. Each state is a couple <S, M>, where S is the current structure of the net, and M is the marking associated to this structure. The root is the initial state $<S_0, M_0>$. The nodes are the set of reachable states. Each two states are linked by an arc labeled with the transition that transforms one state (the first node) to the other state (second node). As in classical Petri Nets, if this tree is finite then many proprieties can decided. In the case of an infinite reachability tree, the analysis will not be complete. In this work, we have realized a small prototype, that can be used to compute reachability tree for some given net, and for some number of level in case of infinite tree. The Figure III.16 shows the first interface of the tool that presents the proposed services in a textual mode. The user must enter the net as a specification in a text to the program.

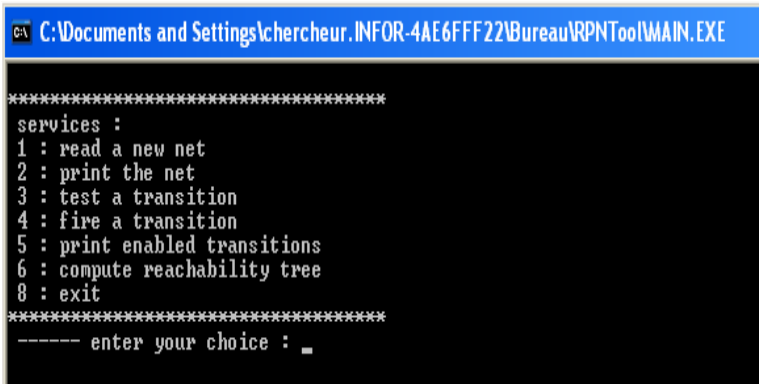


Figure III. 16. Interface of the prototype.

The program can simulate the firing of some transition or it can compute the reachability tree for some given number of levels. As an example, Figure III.18 presents the reachability tree obtained for the net shown on Figure III.17.

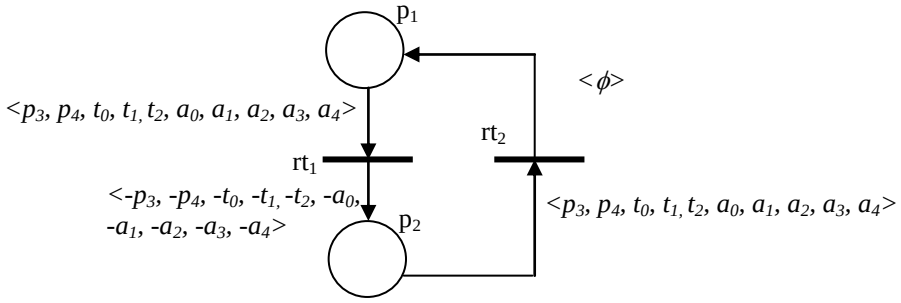


Figure III. 17. Example used as input for the realized prototype

In the example of Figure III.17, the initial marking of the place p_1 is <+p_3, +p_4, +t_0, +t_1, +t_2, +(t_0,p_3), +(p_3,t_1), +(t_1,p_4), +(p_4,t_2), +(t_2,p_2)>. When rt_1 is fired it will add two places: p_3, p_4, three transitions: t_0, t_1, t_2, and the arcs: (t_0,p_3), (p_3,t_1), (t_1,p_4), (p_4,t_2), (t_2,p_2). The new added transitions will not reconfigure the net. After the firing of the transition rt_1, the marking of the place p_2 will be: <-p_3, -p_4, -t_0, -t_1, -t_2, -(t_0,p_3), -(p_3,t_1), -(t_1,p_4), -(p_4,t_2), -(t_2,p_2)>. So when the rt_2 is fired, it will delete the element that rt_1 has added to the net.

Figure III.18 presents the reachability the reachability tree for 36 nodes. The result will be then depicted using the tool **graphviz** [102]. At the current time, the objective of this tool was to simulate the firing of some transition, to follow the dynamic of the net, and to depict some levels of the reachability tree. The more important work requires the implementation of the algorithms that will manipulate this tree to prove and verify the required proprieties. This implementation and more results can be presented in future works.

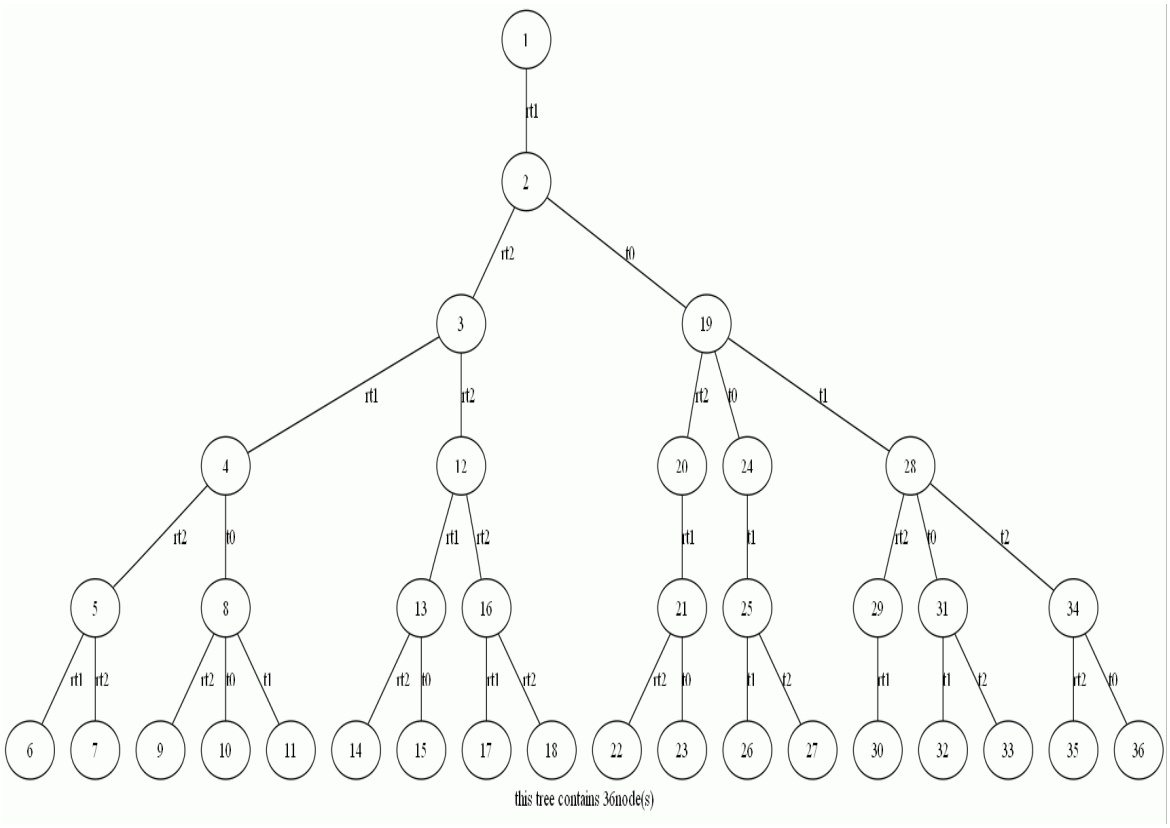


Figure III. 18. Reachability tree

- The second analysis way is to unfold the flexible nets to another low level net for which some analysis tools existed. In our case, we have chosen the unfolding of Flexible Nets toward Dynamic Nets (DN) [5]. The dynamic Nets, as presented in chapter 2, are high level nets, where new transitions can be added to the original nets when some existing transitions are fired in this net. Although the power expressiveness of Dynamic Nets, they impose some constraints on the structure of the net. The high dynamicity of Flexible Nets makes the unfolding complex but possible. We have proposed a transformation technique that transforms the FN into the DN. The objective of this transformation is to profit from the idea that DN can also be unfolded into CPN (Colored Petri Nets). These last one, can be analyzed through the existing tools. The transformation from FN into DN is a formal transformation; this makes it possible to automate the unfolding. This transformation is proved but it is not yet automated. The translation of the Flexible nets into Dynamic nets and the proof of this translation are presented in chapter four.

6 Conclusion

In this chapter, we have presented our contribution at the modeling level in a design process, as a set of extended Petri nets versions. We have presented Labeled Reconfigurable Nets (LRN) [42] as a first and naïve model, and then we have presented its extension to Colored Reconfigurable Nets (CRN) [45]. We have presented for each version some examples of modeling. Studying the shortcomings of these two versions, we have then proposed our last contribution that we consider more mature: "Flexible Nets" (FN) [31], a formalism to specify in a general way all reconfigurable systems. The FN formalism allows the structure of the net to be dynamic during runtime. The net structure can be updated at any time and in high granularity. The nodes of the net's graph can be added or deleted. A marking of a place by negative objects (nodes) will delete these nodes. The marking of places with positive nodes will add these nodes to the structure of the net. We have showed the use of FN to specify some examples specified with other formalisms in the state of the art.

The power of the proposed formalism is that it is more general and gives an intuitive and easy method to specify formally reconfigurable systems. The dynamic structure of a system will be modeled explicitly and naturally in the dynamic structure of the net. We consider that the designer can use this formalism to specify all kinds of reconfigurable systems.

To analyze models, we have proposed tow possible ways: (i) through a simulator tool that simulates the firing of transitions and that depict the reachability tree, (ii) through the unfolding of Flexible Nets specifications into Dynamic Nets models. In the next chapter, we will present the unfolding (or encoding) of Flexible Nets into Dynamic Nets. This encoding will be proved.

Chapter IV: Encoding of Flexible Nets into Dynamic Nets

1 Introduction

Our aim is to offer a way to analyze FN models. As presented in chapter 3, we have proposed that the analysis of FN can be done through the analysis of some equivalent models in CPN (Colored Petri Nets [31]) or PN (Petri Nets [27]). Petri Nets and Colored Petri Nets have been studied for many years and have many automatic verification-tools [33]. To profit from these tools, we must show that there is some correct transformation from the FN formalism (as high level nets) towards CPN (and PN). In this chapter we will present this transformation, and we will present a proof of its correction.

The transformation of FN directly into CPN or PN is a hard task; we propose to prove that the Flexible Nets (FN) models can be encoded into Dynamic Nets (DN) [20]. Dynamic Nets offer the possibility of adding new nets to the original one, when a transition is fired. The problem in this encoding is that our formalism offers more behaviors which are not allowed in DN. In DN, we have some constraints:

Transitions without input places are not allowed (This condition is formulated as the obligation that the set of all mutli-sets $M_{X,C}^{pre}$ is composed of only non empty multi-sets *m*. *m* represent the preset of a transition, see page 4 in [20]);

When adding some nets to the original net, we have not the possibility to modify the input of an existing transition in the original net;

We cannot add a connection between two disconnected existing nodes;

We cannot delete nodes (place, transition or connection).

Authors of [20] gave one example in their paper that presents the adding of a transition. In this context, to encode our model in the DN model, we must prove that all new behaviors of our formalism (adding or deleting some nodes) can be seen as adding of transitions, and then we can encode FN into DN and so into Petri Nets. DN has been encoded into Colored Petri Nets and the encoding is proven in [20].

To allow the reader to follow this encoding, we treat the problems that we meet in such encoding one by one. Every time, when we meet some problem, we explain it and we give how we have resolved it. Firstly, we try to encode the FN into an *indexed DN*. In this indexed DN, we associate to each transition in the DN an explicit name (only to make the model understandable. In Dynamic Net [20], transitions have not explicit names). Names of transitions are bounded like names of places. Names of places and names of transitions are declared together. Indices of transitions and places tell the current reconfiguration (the current firing of a reconfigure transition). For example t^i is the transition which name is t and which is created in the i^{th} reconfiguration of the net. t^{i+1} is the transition which name is t and which is created in the $(i+1)^{th}$ reconfiguration of the net. Even using this indexed DN as an intermediary between FN and DN, important prior transformations are required obtain this indexed DN. This indexed DN is then also transformed into the well known DN.

In this chapter, we present the encoding into Dynamic nets of the principal restructuring behaviors that can be modeled with our formalism: adding a place, adding a transition, adding an arc, deleting a place, deleting a transition, and finally, deleting an arc.

We consider that the adding of a place during runtime is the most complex behavior. To reach the Dynamic Net N_D equivalent to a Flexible Net N_{FN}, we pass through three steps: (i) Transformation of the net to reach a new *equivalent* net N' where the firing of a reconfigure transition will add only new transitions (even if the real behavior does other thing than this) (ii) encoding N' into an indexed Dynamic Nets and the result is $index\_N_D$, and finally, (iii) eliminate the indices and obtain the Dynamic Net N_D. The other behaviors are easier to be encoded into Dynamic nets.

2 Adding a Place in the Flexible Net

In a FN model, a transition can add a place (and so reconfigures the net) when it is fired. To insure this behavior, the transition must have an input place marked <*ap, m0(ap), °ap, ap°, ξ*>, where :

ap: is the name of the place to be added;

m0(ap): is the initial marking of this place;

°*ap*: is the set of input transitions of this place;

ap°: is the set of output transitions of this place;

ξ: is a function that defines the expressions labeling input and output arcs of *ap.*

Figure 1 shows the FN.

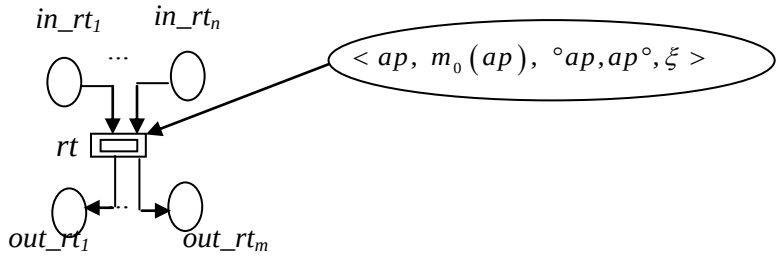


Figure IV.1. Adding a place in the FN model

A transition that can add a place when it is fired, can be seen as a transition with a label <*ap, m0(ap), °ap, ap°, ξ*>. So, the adding of a place can be realized through the firing of a labeled reconfigure transition (as in our work LRN [42]). Figure IV.2 shows the labeled model that does the same behavior. We will work on the model of Figure IV.2.

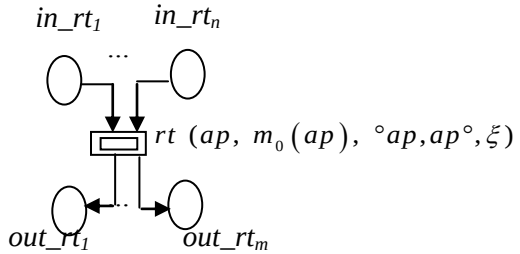


Figure IV.2. Adding a place in the labeled version

2.1 The encoding

In Figure IV.2, we have a transition *rt*, with °*rt* (resp. *rt*°) represents the set of input places (resp. the set of output places) of *rt*. When *rt* is fired, a new place *ap* is added to the net. The new place added will have some input transitions °*ap* and their input expressions: ξ(°*ap*, *ap*), some output transitions *ap*°, and their output expressions: ξ(*ap*, °*ap*), and an initial marking $m_0(ap)$. Transitions in °*ap*, *ap*° are not new transitions but they existed in the original net.

Suppose that:

$^\circ rt = \{in\_rt_1, \ldots, in\_rt_n\}$ is the set of input places of *rt* and $\{x_1, \ldots, x_n\}$ are respectively their associated output expressions so for each i∈1..n, $\xi(in\_rt_i, rt) = x_i$,

$rt^\circ = \{out\_rt_1, \ldots, out\_rt_m\}$ is the set of output places for *rt* and $\{y_1, \ldots, y_m\}$ are respectively their associated input expressions so for each i∈1..n, $\xi(rt, out\_rt_i) = y_i$.

The initial marking of the net is M_0,

The set of transitions in the original net is $T_{FN} = \cup_i \{t_i\} \cup \{rt\}$ where only *rt* is a reconfigure transition,

The set of places in the original net is P_{FN}.

Let consider:

$\{int_1, \ldots, int_k\} \subseteq T_{FN}$ is the set of transitions for which *ap* will be added as an input place with respectively $\{z_1, \ldots, z_k\}$ as a set of expressions such that: for each i∈1..k, $\xi(ap, int_i) = z_i$.

For each transition $int_i \in \{int_1, \ldots, int_k\}$, let °($int_i$) and ($int_i$)° be respectively their input and output places (before starting the transformations).

We denote by int_i^0 the transition int_i at the current time (before any transformation). When *rt* is fired, it adds *ap* to $^\circ(int_i^0)$. In order to make this behavior closer to a Dynamic nets behavior, we propose to simulate this through:

Firstly disabling the int_i^0 transition, (like if this transition does not exist in the net. This effect can be done easily like we will see)

Secondly adding a new transition int_i^1 such that:

$$^\circ(int_i^1) = \{ap\} \cup {}^\circ(int_i^0),$$

$$\text{and } (int_i^1)^\circ = (int_i^0)^\circ \text{ . .....(\textbf{formula I})}$$

Informally, the transition int_i^1 represents the transition int_i^0 with a new input place *ap*. So in the new configuration, int_i^0 will disappear, and we will have a new transition int_i^1 that simulates the transition int_i^0 and which has a new input place *ap*.

Through this first transformation, the adding of a place in the FN model can be seen as the adding of a transition, and this can be seen as a DN behavior. We transform the structure of the FN (Figure 2) towards a new structure (Figure 3) that disables int_i^0, once *rt* is fired. In the Figure IV.3, the place $new\_in_i^0$ (initially marked with a one black token (•)) will disable the transition int_i^0, once the *rt* is fired. In this case, we will have $M_0(new\_in_i^0) = (\bullet)$ and firing *rt* will add a new transition: int_i^1.

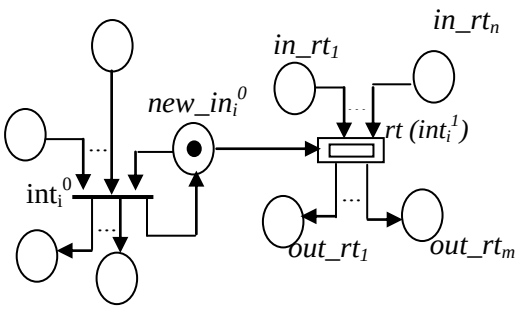


Figure IV.3. Transformation to deal with int_i^0.

Now, the first firing of *rt* is well treated (it disables int_i^0 and it adds int_i^1). But if *rt* will be fired again, we will meet a new problem, because this second firing of *rt* will create a new int_i^2, where the int_i^1 is always enabled (perhaps); we must remember that the original behavior consists to add places and not transitions like this. To cover this problem, we must introduce some mechanism which will always disable the last int_i^j created once *rt* is fired for the (*j+1*) times. This requires the creation of a synchronization place $new\_in_i^j$ marked by a black token (•), every time when *rt* is fired for some j^{th} times. Names of these places can be pulled from a place SYN_in_i of names. The place $new\_in_i^j$ will disable int_i^j when *rt* is fired next time. To insure this synchronization, we firstly add a new place *psyn_in*$_i$, and then we require that another transition $\mathrm{tsyn\_in}_i^j$ (of course, these *indices* are used here to distinguish between occurrences of these transitions, otherwise $\mathrm{tsyn\_in}_i^j$ will appear in the internal net and so no problem of confusion) must be added every time when a new transition int_i^j is added. $\mathrm{tsyn\_in}_i^j$ has as input place the place $new\_in_i^j$ (which is a new name at every firing) and as output place $psyn\_in_i$ (which is added only one time during the translation). Now, adding a place *ap* that will be an input place for some transition int_i^0 (as shown in Figure IV.2) will be considered as adding two transitions ($\mathrm{int}_i^1, \mathrm{tsyn\_in}_i^1$) to the configuration in Figure IV.4. Now, we can correct the above formula (Formula I) to:

$$^\circ(\mathrm{int}_i^1) = \{new\_in_i^1\} \cup \{ap^1\} \cup {}^\circ(\mathrm{int}_i^0) / \{new\_in_i^0\},$$

and $$(\mathrm{int}_i^1)^\circ = \{new\_in_i^1\} \cup (\mathrm{int}_i^0)^\circ / \{new\_in_i^0\} \quad \text{(Formula II)}$$

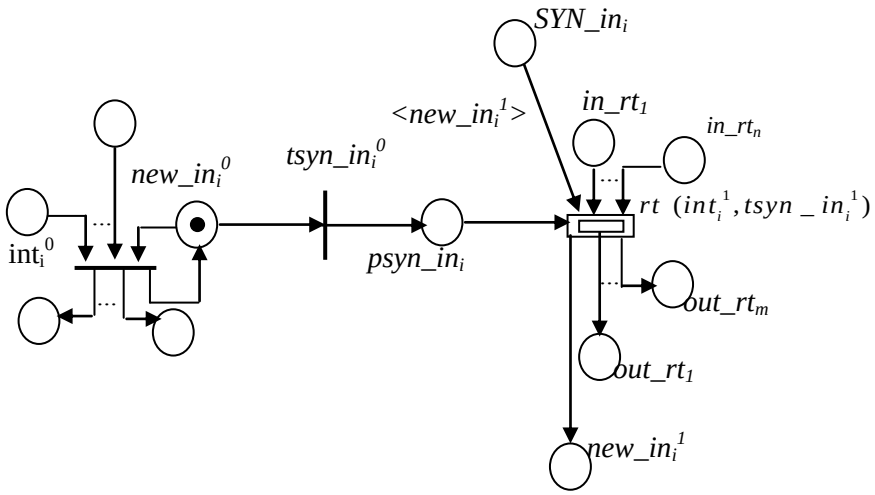


Figure IV.4. Transformation to deal with int_i^1.

In the above section, we have treated the input transition of the added place *ap*. The treatment of the output transition will be similar.

Let consider:

$\{\text{out}_1, \ldots, \text{out}_l\} \subset T_{FN}$ is the set of transitions for which *ap* will be added as an output place with respectively $\{w_1, \ldots, w_l\}$ as expressions, such that: for each $i \in 1..l$, $\xi(out_l, ap)=w_i$.

For each transition $out_i \in \{out_1, \ldots, out_l\}$, let $^\circ\text{out}_i$ and out_i° be respectively their input and output places.

To encode this into DN, we can adopt the same method presented in the above paragraphs. We transform the net to disable out_i^0 once *rt* is fired (by adding a synchronization place $new\_out_i^0$ (where, $M_0(new\_out_i^0) = (\bullet)$), then *rt* adds a new transition out_i^1, such that:

$$(\text{out}_i^1)^\circ = \{ap\} \cup (\text{out}_i^0)^\circ / \{new\_out_i^0\},$$

and

$$^\circ(\text{out}_i^1) = ^\circ(\text{out}_i^0) / \{new\_out_i^0\}. \quad \text{(Formula III)}$$

With the same manner, as in the above, we treat the possible other firings of *rt*. This requires the creation of a synchronization place $new\_out_i^j$ marked by a black token (•), every j^{th} time when *rt* is fired. Names of these places can be pulled from a place SYN_out_i. The place $new\_out_i^j$ will disable out_i^j when *rt* is fired next time. To insure this synchronization, we firstly add a new place $psyn\_out_i$, and then we require that another transition $tsyn\_out_i$' must be added every time when a new out_i^j is added. $tsyn\_out_i^j$ has as input the place $new\_out_i^j$ (which is a new name at every firing) and as output place $psyn\_out_i$ which is a constant.

It is now clear that if *rt* adds a place *ap* such that: $^\circ ap = \{\text{int}_i^0\}_{i=1..k}$ and $ap^\circ = \{\text{out}_i^0\}_{i=1..l}$, this configuration is transformed into an equivalent model in which *rt* adds four transitions $\{\text{int}_i^1, \text{tsyn\_in}_i^1, \text{out}_i^1, \text{tsyn\_out}_i^1\}$.

We conclude that, each transition int_i^j added at the j^{th} firing of *rt*, it will have the same input places of the transition int_i^{j-1} union some new {*ap*} union $\{new\_in_i^j\}$ minus $\{new\_in_i^{j-1}\}$. The output places of int_i^j will be the same output places of the int_i^{j-1} transition union $\{new\_in_i^j\}$ minus $\{new\_in_i^{j-1}\}$.

Formally:

$$^\circ(\text{int}_i^j) = \{ap\} \cup \{new\_in_i^j\} \cup ^\circ(\text{int}_i^{j-1}) / \{new\_in_i^{j-1}\}.$$

and

$$(\text{int}_i^j)^\circ = \{new\_in_i^j\} \cup (\text{int}_i^{j-1})^\circ / \{new\_in_i^{j-1}\}. \quad \text{(Formula IV)}$$

On the other side, for each transition out_i^j added at the j^{th} firing of *rt*, it will have the same input places of the out_i^{j-1} transition union $\{new\_out_i^j\}$ minus $\{new\_out_i^{j-1}\}$. The output places of out_i^j will be the same output places of the out_i^{j-1} transition union {*ap*} union $\{new\_out_i^j\}$ minus $\{new\_out_i^{j-1}\}$.

Formally, $^\circ(\text{out}_i^j) = ^\circ\text{out}_i^{j-1} \cup \{new\_out_i^j\} / \{new\_out_i^{j-1}\}$.

$$\left(\mathrm{out}_i^j\right)^\circ = \{ap\} \cup \left\{new\_out_i^j\right\} \cup \left(\mathrm{out}_i^{j-1}\right)^\circ / \left\{new\_out_i^{j-1}\right\} \quad \text{.....(\textbf{Formula V})}$$

In the following (Figure 5), we present a first encoding of a transformed FN (with one *rt* that adds a place *ap* with *k* input transitions and *l* output transitions) into an indexed Dynamic Nets.

$\upsilon(\mathrm{P_{FN}} \bigcup_{i\in 1..k} \{SYN\_in_i,\ new\_in_i^{\,j}, psyn\_in_i\} \bigcup_{i\in 1..l} \{SYN\_out_i,\ new\_out_i^{\,j}, psyn\_out_i\},$

$\mathrm{T_{FN}} \bigcup_{i\in 1..k} \{\mathrm{tsyn\_in_i}^{\,j}\} \bigcup_{i\in 1..l} \{\mathrm{tsyn\_out_i}^{\,j}\})_{j=0}$

(For $i \in 1..k$,

$\mathrm{int_i}^{\,j}$: $\ldots, new\_in_i^{\,j}(\bullet) \rhd \ldots, new\_in_i^{\,j}(\bullet)$.

$tsyn\_in_i^{\,j} : new\_in_0(\bullet) \rhd psyn\_in_i(\bullet)$.

For $i \in 1..l$,

$\mathrm{out_i}^{\,j}$: $\ldots, new\_out_i^{\,j}(\bullet) \rhd \ldots, new\_out_i^{\,j}(\bullet)$.

$tsyn\_out_i^{\,j} : new\_out_i^{\,j}(\bullet) \rhd psyn\_out_i(\bullet)$.

For each $\mathrm{t_i} \in \mathrm{T_{FN}} / (ap^\circ \cup {}^\circ ap)$,

$\mathrm{t_i}$: $\ldots \rhd \ldots$

$rt : in\_rt_1(x_1), \ldots, in\_rt_n(x_n)$,

$psyn\_int_1(\bullet), \ldots, psyn\_int_k(\bullet)$,

$SYN\_in_1\left(new\_in_1^{\,j+1}\right), \ldots, SYN\_in_k\left(new\_in_k^{\,j+1}\right)$,

$psyn\_out_1(\bullet), \ldots, psyn\_out_l(\bullet)$,

$SYN\_out_l\left(new\_out_1^{\,j+1}\right), \ldots, SYN\_out_l\left(new\_out_l^{\,j+1}\right) \rhd$

$\{\upsilon(\{ap\}, \bigcup_{i\in 1..k} \{\mathrm{int_i}^{\,j}, \mathrm{tsyn\_in_i}^{\,j}\} \bigcup_{i\in 1..l} \{\mathrm{out_1}^{\,j}, \mathrm{tsyn\_out_1}^{\,j}\})_{j=j+1}$

(For $i \in 1..k$,

$\mathrm{int_i}^{\,j}$: ${}^\circ\mathrm{int_i}^{\,j-1} / \left\{new\_in_i^{\,j-1}\right\}, ap(z_i), new\_in_i^{\,j}(\bullet) \rhd \mathrm{int_i}^{\,j-1\circ} / \left\{new\_in_i^{\,j-1}\right\}, new\_in_i^{\,j}(\bullet)$.

$\mathrm{tsyn\_in_i}^{\,j} : new\_in_i^{\,j}(\bullet) \rhd psyn\_in_i(\bullet)$.

For $i \in 1..l$,

$\mathrm{out_i}^{\,j}$: ${}^\circ\mathrm{out_i}^{\,j-1} / \left\{new\_out_i^{\,j-1}\right\}, new\_out_i^{\,j}(\bullet) \rhd {}^\circ\mathrm{out_i}^{\,j-1} / \left\{new\_out_i^{\,j-1}\right\}, ap(\mathrm{w_i}), new\_out_i^{\,j}(\bullet)$.

$\mathrm{tsyn\_out_i}^{\,j} : new\_out_i^{\,j}(\bullet) \rhd psyn\_out_i(\bullet)$.

$), m_0(ap)$

$\}$

$out\_rt_1(y_1), \ldots, out\_rt_m(y_m)$,

$new\_in_1^{\,j}(\bullet), \ldots, new\_in_k^{\,j}(\bullet)$,

$new\_out_1^{\,j}(\bullet), \ldots, new\_out_l^{\,j}(\bullet)$.

$\}$,

M_0,

$new\_in_1^{\,j}(\bullet), \ldots, new\_in_k^{\,j}(\bullet)$,

$SYN\_in_1(\infty), \ldots, SYN\_in_k(\infty)$,

$new\_out_1^{\,j}(\bullet), \ldots, new\_out_l^{\,j}(\bullet)$,

$SYN\_out_1(\infty), \ldots, SYN\_out_l(\infty)$.

)

Figure IV.5. The indexed Dynamic Net

Now, to obtain the Dynamic net, we must eliminate these indices. We see that each transition $\mathrm{int_i}^{\,j}$ ($j \in 1..n$) will have as preset : ${}^\circ(\mathrm{int_i}) \cup \left\{ap^1(b), \ldots, ap^j(b)\right\} \cup \{new\_in^j(\cdot)\}$, and as postset : $(\mathrm{int_i})^\circ \cup \{new\_in^j(\cdot)\}$. So the most problem that we must resolve is the increasing in the set $\left\{ap^1(b), \ldots, ap^j(b)\right\}$ in each firing. These places must be saved from some firing *j* to the next firing *j+1* (of *rt*). To encode this into DN, we must consider that these places and

their associated presets or postsets are always saved in some places (names of these places are always prefixed by *p_ap*). These places are input-output places for the *rt* transitions. These places have a particular type: *Arrays*. Tokens of these places are arrays $\{X_i\}$. Each token is an array of couple $\{(ap^j, z_i)\}_j$, where ap^j will be the name of a place and z_i an expression. We add also an input-output place *index_ap* (with integer as its type) which will index *ap* occurrences. The initial marking of *index_ap* is 1.

The last necessary transformation, to reach the Dynamic net, is now:

- We add *index_ap(ind)* to the preset of *rt* and *index_ap(ind+1)* to its postset
- We add an input-output place $p\_ap\_in_i$ to *rt* with the initial marking $\{(ap^1, z_i)\}_{i=1...k}$
- We add an input place $p\_ap\_out_i$ to *rt* with the initial marking $\{(ap^1, w_i)\}_{i=1..l}$
- We add to the preset of *rt*: $p\_ap\_in_i(X_i)$. The X_i is evaluated to the token stored in $p\_ap\_in_i$ of the form : $<(ap^1, z_i), \ldots, (ap^j, z_i)>,\ i = 1..k$
- We add to the preset of *rt*: $p\_ap\_out_i(Y_i)$. The Y_i is evaluated to the token stored in $p\_ap\_in_i$ of the form : $<(ap^0, w_i), \ldots, (ap^j, w_i)>,\ i = 1..l$
- We add to the postset of $rt : p\_ap\_in_i(X_i, (ap^{ind}, z_i)),\ \ i = 1..k$
- We add to the postset of $rt : p\_ap\_out_i(Y_i, (ap^{ind}, w_i)),\ \ i = 1..l$

For each transition int_i (for which *ap* will be added as an input place, with z_i as input expression), we do the following:

- In the preset of int_i we add: $\{ap^j(z_j) / (ap^j, z_j) \in X_i\}_{j=1..}$

For each transition out_i (for which *ap* will be added as an output place, with w_i as output expression), we do the following:

- In the preset of out_i we put: $\{ap^j(w_j) / (ap^j, w_j) \in Y_i\}_{j=1..}$

Finally, it is clear that the encoding now depends on the marking of the Net. We allow the *rt* to create *n_ap* new places *ap*. To insure this, we can add a new input place *max* to *rt* initially marked *n_ap* ($M_0(max) = n\_ap$). We add *max(•)* in the preset of *rt*. The created places $\{ap^j\}_j$ must be initialized now, and to insure this we add a transition in the internal net called init_mark, such that: $\mathrm{init\_mark} : ap(x) \triangleright ap^{ind}(x)$.

The correct encoding will be as follows (Figure 6):

$$\upsilon(P_{FN} \bigcup_{i \in 1..k} \{SYN\_in_i,,\ new\_in_i, psyn\_in_i\} \bigcup_{i \in 1..l} \{SYN\_out_i,,\ new\_out_i, psyn\_out_i\} \cup$$
$$\{max\} \cup \{p\_ap\_in_1,\ \ldots, p\_ap\_in_k, p\_ap\_out_1, \ldots, p\_ap\_out_l\}$$
$$\cup \{index\_ap\}, T_{FN} \bigcup_{i=1..k} \{tsyn\_in_i\} \bigcup_{i \in 1..l} \{tsyn\_out_i\})$$
(For $i \in 1..k$,
int_i : $\ldots, new\_in_i(\bullet) \rhd \ldots, new\_in_i(\bullet)$.
$tsyn\_in_i : new\_in_i(\bullet) \rhd psyn\_in_i(\bullet)$.
For $i \in 1..l$,
out_i : $\ldots, new\_out_i(\bullet) \rhd \ldots, new\_out_i(\bullet)$.
$tsyn\_out_i : new\_out_i(\bullet) \rhd psyn\_out_i(\bullet)$.
For each $t_i \in T_{FN} / (ap^\circ \cup {}^\circ ap)$,
t_i : $\ldots \rhd \ldots$
$rt : in\_rt_1(x_1), \ldots, in\_rt_n(x_n)$,
$psyn\_int_1(\bullet), \ldots, psyn\_int_k(\bullet)$,
$SYN\_in_1(new\_in_1), \ldots, SYN\_in_k(new\_in_k)$,
$psyn\_out_1(\bullet), \ldots, psyn\_out_l(\bullet)$,
$SYN\_out_1(new\_out_1), \ldots, SYN\_out_l(new\_out_l)$,
$p\_ap\_in_1(X_1), \ldots, p\_ap\_in_k(X_k)$,
$p\_ap\_out_1(Y_1), \ldots, p\_ap\_out_l(Y_l)$,
$max(\bullet)$, index_ap(ind) $\rhd$
$\{\upsilon(\{ap\}, \bigcup_{i \in 1..k} \{tsyn\_in_i\} \bigcup_{i=1..l} \{tsyn\_out_i\} \cup \{init\_mark\})$
(For $i \in 1..k$,
int_i : $^\circ int_i, \{ap^j(z_j) / (ap^j, z_j) \in X_i\}_{j=1..n\_ap}, new\_in_i(\bullet) \rhd int_i{}^\circ, new\_in_i(\bullet)$.
$tsyn\_in_i : new\_in_i(\bullet) \rhd psyn\_in_i(\bullet)$.
For $i \in 1..l$, out_i :
$^\circ out_i, new\_out_i(\bullet) \rhd {}^\circ out_i, \{ap^j(w_i) / (ap^j, w_i) \in Y_i\}_{j=1..n\_ap}, ap(w_i), new\_out_i(\bullet)$.
$tsyn\_out_i : new\_out_i^j(\bullet) \rhd psyn\_out_i(\bullet)$.
init_mark : $ap(x) \rhd ap^{ind}(x)$.
$m_0(ap)\}$
)}
$out\_rt_1(y_1), \ldots, ou\_rt_m(y_m)$,
$new\_in_1(\bullet), \ldots, new\_in_k(\bullet)$,
$new\_out_1(\bullet), \ldots, new\_out_l(\bullet))$,
$p\_ap\_in_1(X_1, (ap^{ind}, z_1)), \ldots, p\_ap\_in_k(X_k, (ap^{ind}, z_k))$,
$p\_ap\_out_1(Y_1, (ap^{ind}, w_1)), \ldots, p\_ap\_out_l(Y_l, (ap^{ind}, w_l))$,
index_ap(ind + 1)
},
$M_0, max(n\_ap)$,
$p\_ap\_in_1((ap^1, x_1)), \ldots, p\_ap\_in_k((ap^1, x_k))$,
$p\_ap\_out_1((ap^1, w_1)), \ldots, p\_ap\_out_l((ap^1, w_l))$,
$new\_in_1(\bullet), \ldots, new\_in_k(\bullet)$,
$SYN\_in_1(new\_in_1^1 + \ldots + new\_in_{n\_ap}{}^1), \ldots, SYN\_in_k(new\_in_1{}^k + \ldots + new\_in_{n\_ap}{}^k)$,
$new\_out_1(\bullet), \ldots, new\_out_l(\bullet)$,
$SYN\_out_1(new\_out_1^1 + \ldots + new\_out_{n\_ap}{}^1), \ldots, SYN\_out_l(new\_out_1{}^l + \ldots + new\_out_{n\_ap}{}^l)$,
$index\_ap(1)$,
)

Figure IV.6. The final Dynamic net

2.2 Example of an encoding

Let consider the example of Figure IV.7. In this example, we consider that all places have the same type *T*. *x* is a variable in *T*. *b* is a constant in *T*. The initial marking of the net is $M_0=p_3(b+b)$. In this FN, the firing of *rt* will add a new place *ap* with an initial markin <*a*>. this place has an input transition *int* with an expression <*x*> and an output transition *out* with an expression <*b*>.

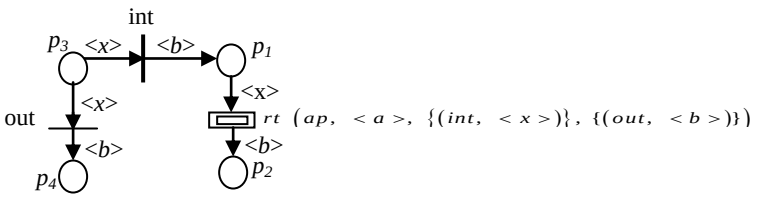


Figure IV.7. Example of a Flexible Net.

Applying the transformation rules presented above, we will obtain the net presented in the Figure IV.8.

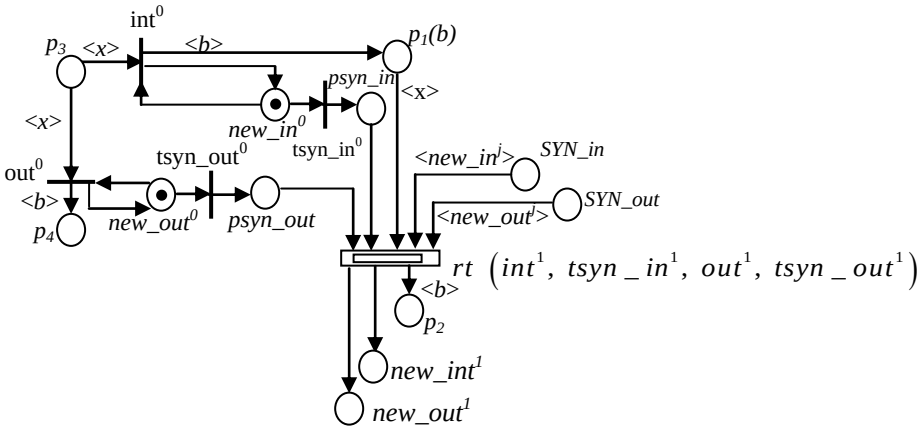


Figure IV.8. Transformation of the example.

In Figure IV.8, int^1 is the transition: $p_3(x), ap(x), new\_int^1(\bullet) \triangleright p_1(b), new\_int^1(\bullet)$, so we can write it

$\text{Int}^1 : {}^\circ(\text{int}^0) / \{new\_in^0\}, ap(\text{b}), new\_in^1(\bullet) \triangleright (\text{int}^0)^\circ / \{new\_in^0\}, new\_in^1(\bullet)$.

tsyn_in^1 is the transition: $new\_in^1(\bullet) \triangleright psyn\_in(\bullet)$.

out^1 is the transition: $p_3(x), new\_out^1(\bullet) \triangleright p_4(b), ap(b), new\_out^1(\bullet)$.

So we can write it :

$\text{out}^1 : {}^\circ(\text{out}^0) / \{new\_out^0\}, new\_out^1(\bullet) \triangleright (\text{out}^0)^\circ / \{new\_out^0\}, ap(b), new\_out^1(\bullet)$.

tsyn_out^1 is the transition: $new\_out^1(\bullet) \triangleright psyn\_out(\bullet)$.

The encoding of this net is the following (Figure 9):

$\upsilon(\{p_1, p_2, p_3, p_4, SYN\_in, new\_in^j, psyn\_in, SYN\_out,, new\_out^j, psyn\_out,\},$

$\{rt, \text{int}^j, \text{out}^j, \text{tsyn\_in}, \text{tsyn\_out}\})_{j=0}$ // *external net initially 0 firing of rt*

$(\{\text{int}^j : p_3(x), new\_in^j(\bullet) \triangleright p_1(b)), new\_in^j(\bullet).$

$\text{tsyn\_in} : new\_in^j(\bullet) \triangleright psyn\_in(\bullet).$

$\text{out}^j : p_3(x), new\_out^j(\bullet) \triangleright p_4(b), new\_out^j(\bullet).$

$\text{tsyn\_out} : new\_out^j(\bullet) \triangleright psyn\_out(\bullet).$

$rt : p_1(x), psyn\_in(\bullet), SYN\_in(new\_in^{j+1}), psyn\_out(\bullet), SYN\_out(new\_out^{j+1}) \triangleright$

$\{\upsilon(\{ap\}, \{\text{int}^j, \text{out}^j, \text{tsyn\_in'}, \text{tsyn\_out'}\})_{j=j+1}$ // *internal net j^{th} firing of rt*

$(\{\text{int}^j : {}^\circ(\text{int}^{j-1}) / \{new\_in^{j-1}\}, ap(x), new\_in^j(\bullet) \triangleright (\text{int}^{j-1})^\circ / \{new\_in^{j-1}\}, new\_in^j(\bullet),$

$\text{tsyn\_in'} : new\_in^j(\bullet) \triangleright psyn\_in(\bullet).$

$\text{out}^j : {}^\circ(\text{out}^{j-1}) / \{new\_out^{j-1}\}, new\_out^j(\bullet) \triangleright (\text{out}^{j-1})^\circ / \{new\_out^{j-1}\}, ap(b), new\_out^j(\bullet),$

$\text{tsyn\_out'} : new\_out^j(\bullet) \triangleright psyn\_out(\bullet).$

},

$ap(a)$ // *initial marking of internal net*

)

}, $new\_in(\bullet), new\_out(\bullet), p_2(b).$ // *other elements in the postset of rt*

},

$P_1(b+b), new\_in^0(\bullet), SYN\_in(\infty), new\_out^0(\bullet), SYN\_out(\infty).$ // *initial marking of external net*

)

Figure IV.9. Indexed DN for the example.

We eliminate indices and we take for example *n_ap=3*. We will have the DN (Figure 10) :

$\upsilon(\{p_1,\ p_2,\ p_3,\ p_4, SYN\_in, new\_in, psyn\_in,\ SYN\_out, new\_out,\ psyn\_out,$
$max,\ p\_ap\_in,\ p\_ap\_out,\ index\_ap\},$
$\{rt,\ \text{int},\ \text{out},\ \text{tsyn\_in},\ \text{tsyn\_out}\})$
$(\{\text{int} : p_3(x), new\_in(\bullet) \rhd p_1(b), new\_in(\bullet).$
$\text{tsyn\_in} : new\_in(\bullet) \rhd psyn\_in(\bullet).$
$\text{out} : p_3(x), new\_out(\bullet) \rhd p_4(b), new\_out(\bullet).$
$\text{tsyn\_out} : new\_out(\bullet) \rhd psyn\_out(\bullet).$
$rt : p_1(x), psyn\_in(\bullet), SYN\_in(new\_in),\quad psyn\_out(\bullet), SYN\_out(new\_out),$
$\mathbf{p\_ap\_in}(\mathbf{X}),\ \mathbf{p\_ap\_out}(\mathbf{Y}),\ \mathbf{max}(\bullet),\ \text{index\_ap(ind)} \rhd$
$\{n(\{ap\},\ \{\text{int},\ \text{out},\ \text{tsyn\_in},\ \text{tsyn\_out}\})$
$(\{\text{int} : p_3(x), \{ap^j(x) / (ap^j, x) \in X\}_{j=1..3}, new\_in(\bullet) \rhd p_1(b), new\_in(\bullet).$
$\text{tsyn\_in} : new\_in(\bullet) \rhd psyn\_in(\bullet).$
$\text{out} : p_3(x), new\_out(\bullet) \rhd p_4(b), \{ap^j(w) / \ (ap^j, w) \in Y\}_{j=1..3}, new\_out(\bullet).$
$\text{tsyn\_out} : new\_out(\bullet) \rhd psyn\_out(\bullet)$
$\text{init\_mark} : ap(x) \rhd ap^{ind}(x)$ // to initialise the marking of ap^{ind}
$\},$
$ap(a))$ // to initial marking of ap
$\},$
$p_2(b), new\_in(\bullet), new\_out(\bullet), \text{index\_ap(ind + 1)},$
$p\_ap\_in(X,\ (ap^{ind+1},\ z)),\ p\_ap\_out(Y,\ (ap^{ind+1}, w)).$ // also postset of rt
$\},$
$P_1(b + b), max(\bullet + \bullet + \bullet),$
$p\_ap\_in((ap^1, x)),\ p\_ap\_out((ap^1, w)),$
$new\_in(\bullet), SYN\_in\ (new\_in^1 + new\_in^2 + new\_in^3),$
$new\_out(\bullet), SYN\_out\ (new\_out^1 + new\_out^2 + new\_out^3).$
$index\_ap(1)$ // marking of the external net
$)$

Figure IV.10. DN for the example.

2.3 Simulation on the example

In the FN of Figure IV.7 (or Figure IV.11): The initial marking is: $s_0 =< p_3(b + b) >$.

Let fire int twice:

$$s_0 \rightarrow^{\text{int}} s_1 =< p_3(b) > \rightarrow^{\text{int}} s_2 =< p_1(b + b) >.$$

Now let fire *rt* twice (see Figure IV.12):

$$s_2 \rightarrow^{\text{rt}} s_3 =< p_2(b), ap^1(a) > \rightarrow^{\text{rt}} s_4 =< p_2(b + b), ap^1(a),\ ap^2(a) >.$$

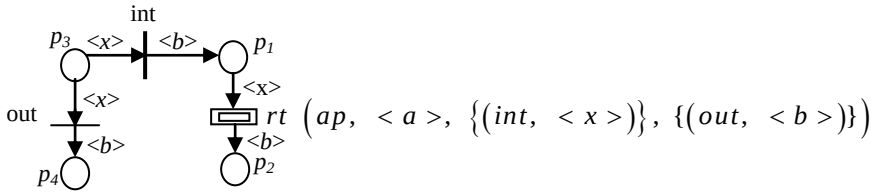


Figure IV.11. The model before firing rt.

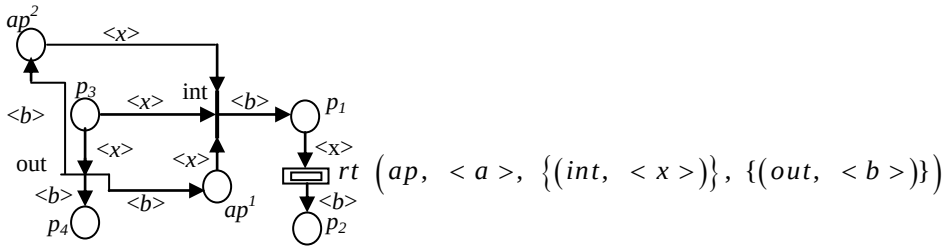


Figure IV.12. The model after firing rt twice.

In the Dynamic Net :

$$s_0' =< DN,\ \{p_3(b+b), max(3),\ p\_ap\_in((ap^1, x)),\ p\_ap\_out((ap^1, w)),$$
$$new\_in^0(\bullet), SYN\_in(new\_in^1 + new\_in^2 + new\_in^3), new\_out^0(\bullet),$$
$$SYN\_out(new\_out^1 + new\_out^2 + new\_out^3), index\_ap(1)\} >$$

Firing int twice:

$$s_0' =< DN,\ \{p_3(b+b), max(3),\ p\_ap\_in((ap^1, x)),\ p\_ap\_out((ap^1, w)),$$
$$new\_in^0(\bullet), SYN\_in\ (new\_in^1 + new\_in^2 + new\_in^3),$$
$$new\_out^0(\bullet), SYN\_out\ (new\_out^1 + new\_out^2 + new\_out^3), index(1)\} >$$

$$\rightarrow^{int} s_1' =< DN,\ \{p_3(b), p_1(b), max(3),\ p\_ap\_in((ap^1, x)),\ p\_ap\_out((ap^1, w)),$$
$$new\_in^0(\bullet), SYN\_in(new\_in^1 + new\_in^2 + new\_in^3),$$
$$new\_out^0(\bullet), SYN\_out(new\_out^1 + new\_out^2 + new\_out^3), index(1)\} >$$

$$\rightarrow^{int} s_2' =< DN,\ \{p_1(b+b), max(3),\ p\_ap\_in((ap^1, x)),\ p\_ap\_out((ap^1, w)),$$
$$new\_in^0(\bullet), SYN\_in(new\_in^1 + new\_in^2 + new\_in^3),$$
$$new\_out^0(\bullet), SYN\_out(new\_out^1 + new\_out^2 + new\_out^3), index(1)\} >$$

Firing *rt*: before firing *rt*, we must fire tsyn_in,

$$\rightarrow^{tsyn\_in,\ tsyn\_out} s_3' =< DN,\ \{p_1(b+b), max(3),$$
$$p\_ap\_in((ap^1, x)),\ p\_ap\_out((ap^1, w)),$$
$$psyn\_in(\bullet), SYN\_in(new\_in^1 + new\_in^2 + new\_in^3),$$
$$psyn\_out(\bullet), SYN\_out(new\_out^1 + new\_out^2 + new\_out^3), index(1)\} >$$

$$\rightarrow^{rt} s_4 =< DN_T \cup \{int^1,\ tsyn\_in^1,\ out^1, tsyn\_out^1,\ init\_mark^1\},$$
$$\{p_1(b), p_2(b), max(2), p\_ap\_in((ap^1,x),(ap^2,x)),$$
$$p\_ap\_out((ap^1,w),(ap^2,w)), SYN\_in(new\_in^2 + new\_in^3),$$
$$SYN\_out\ (new\_out^2 + new\_out^3), new\_in^1(\bullet), new\_in^1(\bullet), index(2), ap(a)\} >$$

$$\rightarrow^{init\_mark^1} s_4' =< DN_T \cup \{int^1,\ tsyn\_in^1, out^1,\ tsyn\_out^1, init\_mark^1\},$$
$$\{p_1(b), p_2(b), max(2), p\_ap\_in((ap^1,x),(ap^2,x)),\ p\_ap\_out((ap^1,w),(ap^2,w)),$$
$$SYN\_in(new\_in^2 + new\_in^3), SYN\_out\ (new\_out^2 + new\_out^3),$$
$$new\_in^1(\bullet), new\_in^1(\bullet), ap^1(a), index(2)\} >$$

$$\rightarrow^{rt} s_5' =< DN \cup \{int^1,\ tsyn\_in^1, out^1,\ tsyn\_out^1, int^2,\ init\_mark^1,\ out^2,\ init\_mark^2\},$$
$$\{p_2(b+b), max(1), p\_ap\_in((ap^1,x),(ap^2,x),(ap^3,x)),$$
$$p\_ap\_out((ap^1,w),(ap^2,w),(ap^3,w)), psyn\_in(\bullet), SYN\_in(new\_in^3),$$
$$SYN\_out(new\_out^3), ap^1(a),\ ap(a), new\_in^2(\bullet), new\_in^2(\bullet), index(3)\} >$$

$$\rightarrow^{init\_mark^2} s_6' =< DN \cup \{int^1,\ out^1,\ int^2,\ init\_mark^1,\ out^2,\ init\_mark^2\},\ \{p_2(b+b), max(1),$$
$$p\_ap\_in((ap^1,x),(ap^2,x),(ap^3,x)),\ p\_ap\_out((ap^1,w),(ap^2,w),(ap^3,w)),$$
$$psyn\_in(\bullet), SYN\_in(new\_in^3), SYN\_out(new\_out^3),$$
$$ap^1(a),\ ap^2(a), new\_in^2(\bullet), new\_in^2(\bullet), index(3)\} >$$

We see in this simulation that:

In the FN model: $s_0 \rightarrow^{int} s_1 \rightarrow^{int} s_2 \rightarrow^{rt} s_3 \rightarrow^{rt} s_4$

In the DN model:

$s_0' \rightarrow^{int} s_1' \rightarrow^{int} s_2' \rightarrow^{tsyn\_in,\ tsyn\_out} s_3' \rightarrow^{rt} s_4' \rightarrow^{init\_mark^1} s_5' \rightarrow^{rt} s_6' \rightarrow^{init\_mark^2} s_7'$

The set $\{(s_{0,}s_0'),\ (s_{1,}s_1'),\ (s_{2,}s_2'),\ (s_{3,}s_5'),\ (s_{4,}s_7')\}$ contains pairs of states that represent equivalent markings. This set can be a subset of a ***bisimulation*** between the two models.

2.4 Correction of the encoding

Let *N* be a Flexible Net and N_D its encoding as a Dynamic Net. Let *rt* be the reconfigure transition in *N*, let $< ap, m_0(ap), {}^\circ ap, ap^\circ, \xi >$ be the label of *rt*. ${}^\circ ap = \{int_i\}_{i=1..n}$ and $ap^\circ = \{out_i\}_{i=1..n}$. Let *rt* be the transition that creates internal net in N_D (and so it encodes the *rt* in *N*), let $\{int_i^j\}_{i=1..n;\ j=0..,n\_ap}$ be the set of transitions that encode the set ${}^\circ ap$, and let $\{out_i^j\}_{i=1..m,\ j=0,..,n\_ap}$ be the set of transitions that encode the set ap°. We will denote by $S_N(p)$ the marking of a place $p \in N$. We will denote by $S_{DN}(p)$ the marking of a place $p \in N_D$.

We consider that the encoding of *N* into N_D can be seen as a transformation ψ that associates to each name in *N* an image (some name) in N_D. This transformation is defined as following:

- $\psi(rt) =< rt, init\_mark^s >$. $init\_mark^s$ is the only firable transition in the set $\{int\_mark^j\}_{j=1,..,n\_ap}$,

- For all $p \in N / \{ap\}$, $\psi(p) = p$,
- For each ap^j created at the j^{th} firing of *rt*, $\psi(ap^j) = ap^j$,
- For each $t \in N / (\{rt\} \cup \{^\circ ap\} \cup \{ap^\circ\})$, $\psi(t) = t$.
- For i=1..*k*, for each $int_i \in {}^\circ ap, \psi(int_i) = int_i^s$. int_i^s is the only firable transition in the set $\{int_i^j\}_{i=1..n; j=0,..,n\_ap}$,
- For i=1..*l*, for each $out_i \in ap^\circ, \psi(out_i) = int_i^s$. out_i^s is the only firable transition in the set $\{out_i^j\}_{i=1..n; j=0,..,n\_ap}$.

Semantics of the two formalisms

We propose two LTS (Labeled Transition Systems) S_N, S_{ND} that represent the operational semantics of the two formalisms.

$S_N = (St_N, Lbl_N, \rightarrow_N)$ where:

St_N: is the set of reachable markings $\{s_N\}$ for *N*,

Lbl_N is a labeling function, $Lbl_N : N_T \rightarrow Names$. For each $t \in N_T, Lbl_N(t) = t$.

$\rightarrow_N : St_N \rightarrow^a St_N$; $a \in Lbl_N(N_T)$.

$S_{ND} = (St_{ND}, Lbl_{ND}, \rightarrow_{ND})$ where :

St_{ND} is the set of reachable markings $\{s_{ND}\}$ for N_D.

Lbl_{ND}: $N_{DT} \rightarrow Names$. For each $t \in N_{DT}$, such that there is $t' \in N_T$ and $\psi(t') = t$, $Lbl_{ND}(t) = t$ otherwise $Lbl_{ND}(t) = \tau$ (denoting non observable action),

$\rightarrow_{ND} : St_{ND} \rightarrow^a St_{ND}$ $a \in Lbl_{ND}(N_{DT})$.

Equivalency between states:

A state $s_N \in St_N$ is **equivalent** to a state $s_N \in St_{DN}$, and we write it $s_N \approx s_{ND}$ iff :

For each $p \in N$, we have : $s_N(p) = s_{ND}(\psi(p))$

Proposition (Bisimilarity between a FN and its encoding as a DN):

Let *N* be an FN that contain a reconfigure transition that adds a place and N_D its encoding. Let $S_N = (St_N, Lbl_N, \rightarrow_N)$ and $S_{ND} = (St_{ND}, Lbl_{ND}, \rightarrow_{ND})$ their two LTSs respectively. We denote by τ^* a sequence of labels (eventually empty), denoting a sequence of non-observable actions.

- For each two states s_N and s_{ND}, such that $s_N \approx s_{ND}$, we have

$s_N \rightarrow^t s_N'$ iff $s_{ND} \rightarrow^{\tau^* \psi(t)} s_{ND}'$ such that $s_N' \approx s_{ND}'$

- For each two states s_N and s_{ND}, such that $s_N \approx s_{ND}$, we have

$s_{ND} \rightarrow \tau^{*\psi(t)} s_{ND}'$ iff $s_N \rightarrow^t s_N'$ such that $s_N \approx s_{ND}$

Proof:

We present the proof of the second equivalency. We will adopt an induction method to prove the bisimilarity between the two models.

Proof 1:

Firstly, we prove the equivalence between the two initial states s_N^0 and s_{ND}^0.

We have $s_N^0 \approx s_{ND}^0$ because:

- *ap* is not yet created
- $s_N^0(p) = s_{ND}^0(\psi(p))$. For all $p \in N$, because the encoding as presented doesn't modify the marking of places of the FN.
- $\psi(\text{int}_i) = \{\text{int}_i^0\}, i = 1..k$;
- $\psi(\text{out}_i) = \{\text{out}_i^0\}, \ i = 1..l$;

Now, we prove: $s_{ND}^0 \rightarrow^{t*T(t)} s_{ND}^1$ iff $s_N^0 \rightarrow^t s_N^1$ such that $s_N^1 \approx s_{ND}^1$

Firing *rt*:

In the dynamic net:

In N_D, to fire *rt*, the silent transitions $\{\text{tsyn\_in}_i\}_{i=1..k} \cup \{\text{tsyn\_out}_i\}_{i=1..l}$ must be fired first. This will make $\{\text{out}_i^0\}_{i=1..k;}$ $\{\text{int}_i^0\}_{i=1..l;}$ never firable after. Now firing *rt* will do the following:

- For each $p \in {}^\circ rt$, $s_{DN}^1(p) = s_{DN}^0(p) - \xi(p, rt)$.
- For each $p \in rt^\circ$, $s_{DN}^1(p) = s_{DN}^0(p) + \xi(rt, p)$.
- A place *ap* marked $m_0(ap)$ is added.
- New transitions $\{\text{int}_i^1\}_{i=1..k}$ are added with ap^1 as an input places.
- New transitions $\{\text{out}_i^1\}_{i=1..l}$ are added with ap^1 as an output places.
- New firable transition init_mark^1 is added with the preset $ap(x)$ and the postset $ap^1(x.)$

Firing init_mark^1 will make $s_{DN}^1(ap^1) = m_0(ap)$

In the Flexible net:

In *N*, firing *rt* adds ap^1. $s_N^1(ap^1) = m_0(ap^1) = s_{DN}^1(ap^1)$.

- For each $p \in {}^\circ rt$, $s_N^1(p) = s_N^0(p) - \xi(p, rt) = s_{DN}^1(p) = s_{DN}^1(\psi(p))$.
- For each $p \in rt^\circ$, $s_N^1(p) = s_N^0(p) + \xi(rt, p) = s_{DN}^1(p) = s_{DN}^1(\psi(p))$.

Also, the firability of $\{\text{int}_i^1\}_{i=1..k}$ in N_{DN} depends only on the firability of $\{\text{int}_i\}_{i=1..k}$ in *N*, because the input-output places $\{new\_in_i^1\}_{i=1..k}$ are marked one and never disturb the firing of $\{\text{int}_i^1\}_{i=1..k}$. Idem for $\{\text{out}_i^1\}_{i=1..k}$.

Firing transitions int_i^0 or out_i^0:

In N_D, firing int_i^0 will update the marking of $^\circ(int_i^0)$ and $(int_i^0)^\circ$ in the same manner that firing int_i will update the marking of $^\circ(int_i)$ and $(int_i)^\circ$ in N. The marking of $\{new\_in_i\}_{i=1..k}$ stills always 1. Idem for out_i^0.

Conclusion: $s_N{}^1 \approx s_{ND}{}^1$

Proof 2: Now, we prove the general case: for all s_N, s_{ND}.

Let consider that N is the net at the s^{th} reconfiguration. In this net, we have s places $\{ap^j\}_{j=1..s}$ which were added as input places for $\{int_i\}_{i=1..k}$, and as output places for $\{out_i\}_{i=1..l}$. Let consider that N_D is the net at the s^{th} reconfiguration. The N_D that satisfies $s_N \approx s_{ND}$ must insure:

- It contains $\{int_i^j\}_{i=1..k;\ j=0..s}$ transitions in which $\{int_i^j\}_{i=1..k;\ j=0..s-1}$ will be never firable, because $\{new\_in_i^s\}_{i=1..k;\ j=0..s}$ will still marked () (0 black token) forever. Only $\{int_i^s\}_{i=1..k}$ are firable. $\psi\{int_i\}_{i=1..k} = \{int_i^s\}_{i=1..k}$. $\{\psi(ap^j)\}_{j=1..s} \in \{^\circ int_i^s\}_{i=1..k}$.
- It contains $\{out_i^j\}_{i=1..k;\ j=0..s}$ transitions in which $\{out_i^j\}_{i=1..k;\ j=0..s-1}$ will be never firable, because $\{new\_out_i^s\}_{i=1..k;\ j=0..s}$ will still marked () forever. Only $\{out_i^s\}_{i=1..l}$ are firable. $\psi\{out_i\}_{i=1..l} = \{out_i^s\}_{i=1..l}$. $\{\psi(ap^j)\}_{j=1..s} \in \{(out_i^s)^\circ\}_{i=1..l}$.
- For all $p \in N$, $s_N(p) = s_{ND}(\psi(p))$.

We want to prove that $s_N \approx s_{ND}$ implies that: $s_{ND} \rightarrow^{\tau^*\psi(t)} s_{ND}'$ iff $s_N \rightarrow^t s_N'$ such that $s_N' \approx s_{ND}'$

Firing *rt*:

In the dynamic net:

In N_D, to fire *rt*, the silent transitions $\{tsyn\_in_i\}_{i=1..k} \cup \{tsyn\_out_i\}_{i=1..l}$ must be fired first. This will make $\{out_i^s\}_{i=1..k} \cup \{int_i^s\}_{i=1..l}$ never firable after. Now firing *rt* will do the following:

- For each $p \in {}^\circ rt$, $s_{DN}{}^{s+1}(p) = s_{DN}{}^s(p) - \xi(p, rt)$.
- For each $p \in rt^\circ$, $s_{DN}{}^{s+1}(p) = s_{DN}{}^s(p) + \xi(rt, p)$.
- A place *ap* marked $m_0(ap)$ is added.
- New transitions $\{int_i^{s+1}\}_{i=1..k}$ are added, with ap^{s+1} as an input places.
- New transitions $\{out_i^{s+1}\}_{i=1..l}$ are added, with ap^{s+1} as an output places.

- New firable transition init_mark^{s+1} is added with the preset *ap(x)* and in the postset $ap^{s+1}(x)$

Firing init_mark^{s+1} will make $s_{DN}{}^{s+1}(ap^1) = m_0(ap)$

In the Flexible net:

In *N*, firing *rt* adds ap^{s+1}. $s_N{}^{s+1}(ap^{s+1}) = m_0(ap) = s_{DN}{}^{s+1}(ap^{s+1}) = s_{DN}{}^{s+1}(\psi(ap^{s+1}))$.

- For each $p \in {}^\circ rt$, $s_N{}^{s+1}(p) = s_N{}^s(p) - \xi(p, rt) = s_{DN}{}^{s+1}(\psi(p))$.
- For each $p \in rt^\circ$, $s_N{}^{s+1}(p) = s_N{}^s(p) + \xi(rt, p) = s_{DN}{}^{s+1}(p) = s_{DN}{}^{s+1}(\psi(p))$.

Also, the firability of $\{\text{int}_i^{s+1}\}_{i=1..k}$ in N_{DN} depends only on the fire-ability of $\{\text{int}_i\}_{i=1..k}$ in *N*, because the input-output places $\{\text{new\_in}_i^{s+1}\}_{i=1..k}$ are marked one and never disturb (while *rt* is not fired) the firing of $\{\text{int}_i^{s+1}\}_{i=1..k}$. Idem for $\{\text{out}_i^{s+1}\}_{i=1..k}$.

Firing transitions int_i^s or out_i^s:

In N_D, firing int_i^s will update the marking of ${}^\circ(\text{int}_i^s)$ and $(\text{int}_i^s)^\circ$ in the same manner that firing int_i will update the marking of ${}^\circ(\text{int}_i)$ and $(\text{int}_i)^\circ$ in *N* at the *s* reconfiguration of the net. The marking of $\{new\_in_i^s\}_{i=1..k}$ stills always 1 (while $\{\text{tsyn\_in}_i^s\}_{i=1..k}$ are not fired). Idem for out_i^s.

Conclusion: $s_N{}^{s+1} \approx s_{ND}{}^{s+1}$

Until now, we have presented the encoding of one behavior (adding a place) into the Dynamic nets, and we have proposed a proof of the equivalence between the Flexible net model and its encoding into Dynamic nets. The rest of this chapter will present the encoding of the other behaviors into Dynamic nets, or into Colored Petri nets (if it is possible).

3 Adding a Transition in the Flexible Net

To add a transition *at*, we must have in the FN some reconfigure transition that when fired it adds this transition (Figure 13).

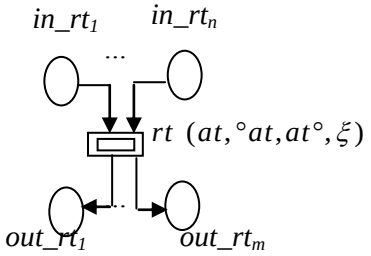


Figure IV.13. Adding a transition

3.1 The encoding

In Figure IV.13, we have a transition *rt*, with °*rt* (resp. *rt*°) represents the input places (resp. the output places) of *rt*. When *rt* is fired, a new transition *at* is added to the net. The new transition added will have some input places °*at* and their input expressions: ξ(°*at*, *at*), and some output transitions *at*°, and their output expressions: ξ(*at*, °*at*). Places in °*at*, and in *at*° are not new places but they existed in the original net. Suppose that: $^{\circ}rt = \{in\_rt_1, \ldots, in\_rt_n\}$ is the set of input places of *rt* and $\{x_1, \ldots, x_n\}$ are respectively their associated output expressions, for each i∈1..n, ξ*(in_rt*$_i$*,rt)=x*$_i$. Suppose that: $rt^{\circ} = \{out\_rt_1, \ldots, out\_rt_m\}$ is the set of output places for *rt* and $\{y_1, \ldots, y_m\}$ are respectively their associated input expressions; for each i∈1..n, ξ*(rt, out_rt*$_i$*)=y*$_i$. The initial marking of the net is M_0, the set of transition is $T_{FN} = \cup_i \{t_i\} \cup \{rt\}$ where only *rt* is a reconfigure transition, the set of places in the original FN is P_{FN}.

Let $\{inp_1, \ldots, inp_k\} \subseteq P_{FN}$ be the places for which *at* will be added as an input transition with respectively $\{z_1, \ldots, z_k\}$ as expressions such that: for each i∈1..k, ξ*(at, inp*$_i$*)=z*$_i$. For each place $inp_i \in \{inp_1, \ldots, inp_k\}$, let °(inp$_i$) and (inp$_i$)° be respectively their input and output transitions (before starting the transformations).

Let $\{outp_1, \ldots, outp_l\} \subseteq P_{FN}$ be the places for which *at* will be added as an output transition with respectively $\{w_1, \ldots, w_l\}$ as expressions such that: for each i∈1..l, ξ*(outp*$_i$*, at)=w*$_i$. For each place $outp_i \in \{outp_1, \ldots, outp_l\}$, let °(outp$_i$) and (outp$_i$)° be respectively their input and output transitions (before starting the transformations).

Dynamic Nets allow the creation of new transition during runtime. When *rt* is fired it adds *at* to $^{\circ}(inp_i^0)$, and to $^{\circ}(inp_i^0)$. This effect can be encoded directly into Dynamic nets. The following specification (Figure 14) does the necessary:

$$
\begin{aligned}
&\upsilon(P_{FN}, T_{FN})\\
&(\\
&\text{For each } t_i \in T_{FN} : t_i : \ \ldots \rhd \ldots\\
&rt : in\_rt_1(x_1), \ \ldots, \ in\_rt_n(x_n)\\
&\qquad\qquad \rhd\\
&\qquad\qquad \{\\
&\qquad\qquad \{at : inp_1(z_1), \ldots, inp_k(z_k) \rhd outp_1(z_1), \ldots, outp_l(z_l)\},\\
&\qquad\qquad out\_rt_1(y_1), \ \ldots, \ ou\_rt_m(y_m),\\
&\qquad\qquad \}\\
&M_0\\
&)
\end{aligned}
$$

Figure IV.14. Adding a transition (The encoding)

In this specification, once *rt* is fired it will create the internal transition *at.*

4 Adding an Arc in the Flexible Net

To add an arc, we must have in the FN some reconfigure transition that when fired it adds this arc (Figure 15).

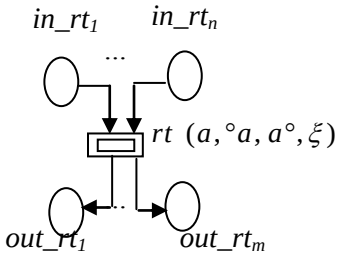


Figure IV.15. Adding an arc

4.1 The encoding

In Figure IV.15, we have a transition *rt*, with °*rt* (resp. *rt*°) represents the input places (resp. the output places) of *rt*. When *rt* is fired, a new arc *a* is added to the net. The new arc added will have an input node (transition or place) °*a* and its input expression: ξ(°*a*, *a*), and one output node (place or transition) *a*°, and its output expression: ξ(*a*, °*a*). °*a* and *a*° are not new nodes but they existed in the original net. Suppose that: $^\circ rt = \{in\_rt_1, \ldots, in\_rt_n\}$ is the set of input places of *rt* and $\{x_1, \ldots, x_n\}$ are respectively their associated output expressions so for each i∈1..n, ξ*(in_rt*$_i$*,rt)=x*$_i$, $rt^\circ = \{out\_rt_1, \ldots, out\_rt_m\}$ is the set of output places for *rt* and $\{y_1, \ldots, y_m\}$ are respectively their associated input expressions so for each i∈1..n, ξ*(rt, out_rt*$_i$*)=y*$_i$. The initial marking of the net is M_0, the set of transition is $T_{FN} = \cup_i \{t_i\} \cup \{rt\}$ where only *rt* is a reconfigure transition, the set of places in the original FN is P_{FN}.

Suppose that *a=(p,t)* is the arc to be added. To encode this behavior in Petri Nets, we propose to decompose this behavior in two behaviors which must be executed in an atomic sequence:

Firstly we will delete the place *p* from the net,

Secondly we add a new place *ap* such that: °*ap*=°*p* and *ap*°=*p*°∪{*t*}.

The place *ap* which has replaced the place *p* in the net has a connection *(ap,t)*, so the arc *a* is added to the net. The encoding of the removing of a place will be presented bellow, and it must never use the present behavior in its encoding, (either else, we will have an infinite loop when encoding a net).

We propose two reconfigure transitions (Figure 16). The first one rt_1 will delete the place *p*, and the second one rt_2 will add the place *ap*.

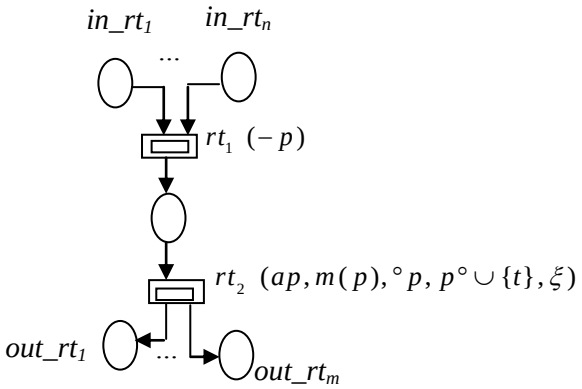


Figure IV.16. Transformation to add an arc *a=(p,t)*

To ensure that the two behaviors will be executed in an atomic sequence, we must oblige that no event can occurs once the sequence starts until the sequence finishes. The first transition must block all transition in the net once it starts its execution. The second transition will liberate all the blocked transitions. We can realize this by adding a set of places (with the number of transitions in the net) which will be initially marked 1 black token, and that must block all transition, once rt_1 is fired. When the transition rt_2 will be fired, it will allow the firing of the blocked transitions. This is an easy synchronization mechanism that can be applied here.

On the Figure IV.16, the behavior of rt_2 can be encoded as done in section 2. The behavior of rt_1 will be presented in the next section.

5 Deleting a Place from the Flexible Net

To delete a place, we must have in the FN some reconfigure transition that when fired it deletes this place (Figure 17).

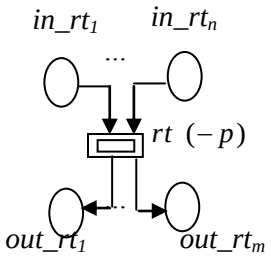


Figure IV.17. deleting a place

5.1 The encoding

Let *p* be a place to be deleted when *rt* is fired. Let °*p* be the set of input transitions for *p* and *p*° the set of output transitions for *p*. The important effect of deleting *p* is on *p*°. Once *rt* is fired, *p* will be deleted and the transitions in *p*° will no never have the input place *p*, and so they will never have a condition depending on the marking of *p*, to be fired. The solution is to

insure that the marking of p never forbid firing of $p°$. One can propose to replace all transitions in $p°=\{t_i\}_{i=1..n}$ by the set of transitions $\{t_i'\}_{i=1..n}$, such that:

for each i=1..n, $°(t_i')=°t_i/\{p\}$.

So the deleting of the place p is done in two steps:

Deleting all transition $p°=\{t_i\}_{i=1..n}$

Adding the set of transitions : $\{t_i'\}_{i=1..n}$, such that:

for each i=1..n, $°t_i'=°t_i/\{p\}$

and $t_i'°=t_i°$

The adding of transition is presented above, and the deleting of a transition is presented in the next section.

6 Deleting a Transition from the Flexible Net

To delete a transition, we must have in the FN some reconfigure transition that when fired, it deletes this transition (Figure 18).

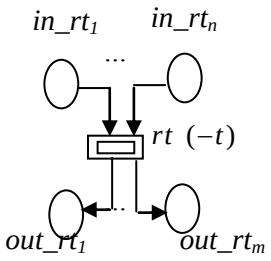


Figure IV.18. Deleting a transition

6.1 The encoding

In Figure IV.18, the transition *rt* deletes the transition *t* when it is fired. When the transition *t* is deleted, this means that the marking of places $°t$ and the marking of places $t°$ will no more be affected by *t*. This can be seen, like if *t* will never been fired. To forbid the firing of *t* once the transition *rt* is fired, one can propose that the firing of *rt* will block *t* by adding an input place *inp* to the transition *t*, such that *inp* is not marked. The Figure IV.18 can be transformed to the Figure IV.19 which will give the same behavior.

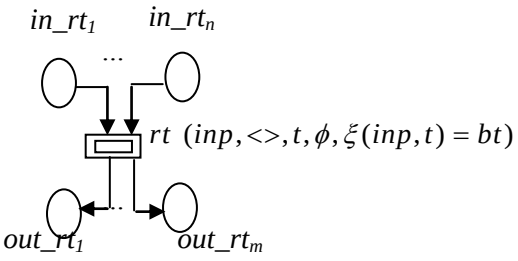


Figure IV.19. Transformation for deleting a transition

In Figure IV.19, *rt* adds the place *inp* not marked as an input place to *t* and the expression of the arc (*inp*,*t*) is one black token. Once added, the place *inp* forbids the firing of *t*, forever.

7 Deleting an Arc from the Flexible Net

To delete an arc, we must have in the FN some reconfigure transition that when fired it deletes this arc (Figure 20).

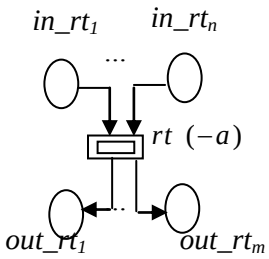


Figure IV.20. Deleting an arc

7.1 The encoding

To delete an arc *a=(p,t)*, such that *p* is a place and *t* is a transition, we can proceed either :

deleting the place *p* from the net, and adding a place *p'* with the same marking of *p*, and where : $°p'=°p$, and $p'°=p°/\{t\}$;

or by deleting the transition *t* from the net, and adding a transition *t'* where : $°t'=°t/\{p\}$, and $p'°=p°$;

To delete an arc *a=(t,p)*, we can proceed with the same manner.

8 Conclusion

When we propose an extended version for a formalism, the most important question is: is the new formalism able to be analyzed? As expressiveness power of the formalism becomes important, as its analyzing becomes difficult. One of the proposed techniques to analyze extended Petri nets was to transform the extended version to a classical one. Low level Petri nets are characterized by their ability to be analyzed, and automatic tools exist for assist the designer to do this analysis. As example, CPN (colored Petri nets) have been proved to be equivalent to some complex representation of Petri nets. Algorithms to unfold CPN to PN are proposed.

In this chapter, we have presented our contribution in the level of analyzing of the proposed extended Petri nets. In this chapter, we have proposed an encoding (translation) from the extended Petri nets (Flexible nets) defined in chapter three to another extended Petri nets (Dynamic nets) proposed in the literature. The choice of Dynamic nets is due to the expressive power of this last one, which makes it the more high level nets that can encode Flexible nets. The choice of Dynamic nets is due also to the possibility of the transformation of these last ones into Colored Petri Nets.

However, even the expressiveness of Dynamic nets as formalism for dynamic systems; we were obliged to apply many transformations on the Flexible nets models before their encoding

into Dynamic nets. In this encoding, we have consider the complex behavior offered in Flexible nets (Adding nodes, deleting nodes), and we have proved that each one of this behavior can be encoded into Dynamic nets, and so into Petri nets. This proves that the Flexible nets save the semantics of Petri nets, and so it is a formal tool that can be analyzed with the same techniques proposed to Petri nets. This result gives to the formalism another advantage, other than making the design and the specification of reconfigurable systems easier and softer.

Conclusion

Conclusion

Reconfigurable systems are systems with a dynamic structure. Their structure changes as they are executed. This class of systems can be found in many domains of our life. Mobile robots used to explore hostile environment, mobile agents used in the internet or in distributed systems, mobile nodes in a mobile wireless networks ... All this systems can be considered as reconfigurable systems. The use of this system is in expansion for many reasons: their efficiency, their abstractions for the designer, their flexibility ...These characteristics make these systems in the kernel of many critical systems: aeronautics, military, medicine, commerce... The design of these systems becomes a critical activity. Their reliability and their correction are crucial. To insure the correction of these systems, formal methods seem to be an adequate solution. Using formal methods, the designer specifies the system in a formal language. A formal language has a well defined syntax, and formal semantics which allows the verification of properties of the designed system. We found in the literature, many formal methods.

Classical formal methods (proposed for classical systems) are well defined and are mature. However, these classical formal methods have not the expressiveness to specify reconfigurable systems. The use of the classical methods makes the designer's task a hard task. Extended versions are proposed to deal with the idea of reconfigurable systems. In the literature, we can find two principal classes: Processes algebra based methods, and state-transition based methods. Processes algebra methods are based on the CSS (Calculus of Communicating Systems) [1] calculus. Extensions for CSS are focused on mobility. The most important extensions that we can find are: π-calculus [2, 3], in which we can specify processes that communicate using channels which can be sent from one process to another, HO-π-calculus [52] is a more reach extension, where processes (here called agents) are mobile. In HO-π-calculus, agents can be sent via channels. The Join calculus [5] is another extension of processes algebra calculi. The characteristic of the Join calculus is that it defines the concept of location. Locations (locality) are important to specify mobility. The introduction of locations in the join calculus makes the calculus more expressive and realistic.

The second class of methods can be found in extensions of Petri nets model. Petri nets are an elegant model for concurrency. With its graphical representation and its formal background, it was used to specify and verify concurrent multi-processes systems. The classical model has not the power of expressiveness to deal with current aspects such as mobility. To take benefits from the power of the model in mobility domains, several works have been proposed. These works try to extend Petri nets with the same ability to specify mobility (and more generally: reconfigurability). We can distinguish between extensions that model mobility in an implicit way (no modification in the structure of the net), or in an explicit way (the net reconfiguration models components mobility).

1 Comparison with Similar works

Research on the use of Petri nets to model systems with dynamic structure has provided some remarkable results. The most important propositions are dedicated to mobile systems and mobile agents. In PrN (Predicate/Transition nets) [25], mobile agents are modeled

through ***tokens***. These agents are transferred by transition firing from an environment to another. In this work, the structure of the net ***does not change***. The agents are represented as token, so this abstraction does not allow representing some complex behavior of this kind of agents. In [24], authors proposed MSPN (Mobile synchronous Petri net) as formalism to model mobile systems and security aspects. They have introduced the notions of nets (an entity) and disjoint locations to ***explicit mobility***. A system is composed of set of localities that can contain nets. To explicit mobility, specific transitions are introduced. Two kinds of specific transitions were proposed: ***new and go***. Firing a go transition moves the net from its locality towards another locality. The destination locality is given through a token in an input place of the go transition. In this work, mobility ***is not also explicit***. Mobility is implicitly modeled by the activation of some nets and the deactivation of other nets, using tokens. ***Migration of an agent is modeled by the deactivation of the net modeling this agent*** in a locality and the activation of the net that represents this same agent in the destination locality. So, this is a kind of simulation of mobility. In nested nets [22], tokens can be Petri nets themselves. This model allows some transition when they are fired ***to create new nets*** in the output places. Nested nets are hierarchic nets where we have different levels of details. Places can contain nets, and these nets can contain also nets as tokens in their places et cetera. So all nets created when a transition is fired are ***contained in places***. So the created nets ***are not in the same level with the first net.*** This formalism is proposed to adaptive workflow systems. In "reconfigurable net" [19], the structure of the net ***is not explicitly changed***. No places or transitions are added in runtime. The key difference with colored Petri nets is that firing transition can ***change names of output places***. Names of places can figure as weight of output arcs. This formalism is proposed to model nets with ***fixed components*** but where connectivity can be changed over time. In [26], PEPA nets are proposed, where mobile code is modeled by expressions of the stochastic process algebra PEPA which play the role of tokens in (stochastic) Petri nets. The Petri net of a PEPA net models the architecture of the net, which is ***a static one***. Mobile Petri nets (MPN) [20] extend colored Petri nets to model mobility. MPN is inspired from join-calculus [5]. The ***output places of transition are dynamic***. The input expression of a transition defines the set of its output places.

In all these formalisms, the structure of the net ***is not changed*** and mobility is modeled implicitly through the net's dynamic. In this model, an important work is required from the modeler to model mobility implicitly. MPN are extended to Dynamic Petri Net (DPN) [20]. Mobility in DPN is modeled explicitly, by adding subnets when transitions are fired. However, the Dynamic Petri nets formalism implies some constraints:

No transition without input places,

Added nets, to the original net, must not modify the input of an existing transition in the original net,

We can't add a connection between two disconnected existing nodes,

and we cannot delete nodes (place, transition or connection).

Through this thesis, we have proposed some extensions to Petri nets that can be used to model mobility (and in general reconfigurable systems). The most important one was the Flexible nets. Flexible Nets is ***more flexible and more expressive*** and doesn't imply constraints on the dynamic of the structure. We consider that Flexible Nets can be used by reconfigurable systems developers with more flexibility than other formalisms. This is due to

the feature that it models ***mobility explicitly*** through mobility of nodes in the Flexible Net. Developers can encode mobile aspects of their system ***directly and explicitly*** in the FN formalism.

The power of Petri nets resides in its verification methods. When extending Petri nets, we reach some formalism with a high expressivity, but the analysis becomes more complex or even impossible. Developers of new formalisms must propose analysis techniques. Mostly, they are proposing some translation (or encoding) of their formalisms into some well-known formalism or approach in modeling domain. Such translation allows the analysis of the new formalism's models using techniques of well-known formalisms. The most famous encoding can be found in the unfolding of Petri nets into automaton to apply model-checking, and then the unfolding of CPN [31] (Colored Petri Nets) into PN [27] (Petri Nets) to analyze some properties that are not analyzed on the CPN directly. We can find other works, in literature. In [33], author authors studied equivalence between the join calculus [5] and different kinds of high level nets. They proved the equivalence between Reconfigurable nets (RN) (an extension version of PN) and the join calculus. This equivalence allows to interpret RN into join calculus and to verify those using join-calculus tools. In [34], Petri nets nets are translated into linear logic programming. This translation can be used to analyze Petri nets using prolog model-checker. Author of [36] encoded Synchronous mobile nets (SMN) [24] into rewriting logic [39]. This encoding allows the use of Maude [37] to verify SMN's specifications.

In this Thesis, we have proposed some methods to analyze models designed into our proposed formalisms. We have presented firstly an automatic verification method that can be used to analyze models. We have realized a tool that can depict the reachability tree of a flexible net, and then some properties can be verified through this reachability tree. A most important work was in the encoding (or unfolding) of flexible nets into Petri nets. In this thesis, we have proposed an encoding of Flexible nets behaviors into Dynamic nets [20]. This encoding was proved to be correct. The advantage of such encoding resides on the possibility to encode Dyanmic nets into CPN (colored Petri Nets). So, Flexible nets can be translated into CPN. Once translated into CPN, Flexible nets can be analyzed using CPN verification tools.

2 Perspectives

As perspectives of the current work, we consider that some important work can be done through two axes. We propose three axes as open domains:

The experimentation of Flexible nets in the modeling of mobile systems: Mobile agents systems, mobile networks, …This modeling work can proof the power of our formalism and shows its shortcomings and so allow us to introduce necessary adaptations;

The work on automatic verification: The realized tool presented in chapter three is not yet completed. The encoding presented in chapter four is formal and proved to be correct; so it is possible to think for an automatic encoding. A third way is to think to encode Flexible nets into rewriting logic. We have proposed an idea [46] to encode an extended Petri nets into a specific Maude (that we have called R-Maude: reconfigurable Maude). We have presented a specification of the R-Maude, but we have not done a complete realization of R-Maude (Only a simulator through a network was realized).

References

References

Process algebra references:

1. R. Milner. "*A Calculus of Communicating Systems*". Number 92 in Lecture Notes in Computer Science. Springer Verlag, 1980.
2. R. Milner, J. Parrow, and D. Walker. "*A calculus of mobile processes*". Information and Computation, 100:1–77, 1992.
3. D. Sangiorgi and D. Walker. "*The π-Calculus: A Theory of Mobile Processes*". Cambridge University Press, 2001.
4. L. Cardelli and A. D. Gordon. "*Mobile Ambient* ", *In Proceedings FoSSaCS'98*, LNCS 1378, pages 140-155. Springer, 1998. Accepted for publication in Theoretical Computer Science.
5. Cédric Fournet and Georges Gonthier. "*The Join Calculus: a Language for Distributed Mobile Programming*". In Applied Semantics. International Summer School, APPSEM 2000, Caminha, Portugal, September 2000, LNCS 2395, pages 268-332, Springer-Verlag. August 2002.
6. Marta Kwiatkowska, Gethin Norman, David Parker, Maria Grazia Vigliotti, "*Probabilistic Mobile Ambients*", Theoretical Computer Science 410, pages : 1272-1303. Elsevier.
7. Abadi, M. and A.D. Gordon. "*A calculus for cryptographic protocols: the spi-calculus*". Proc. of the Fourth ACM Conference on Computer and Communications Security, pages: 36-47, 1997.
8. J. Hillston. "*A Compositional Approach to Performance Modelling"*. Cambridge. University Press, 1996.
9. Baeten, J.C.M. "*Over 30 years of process algebra: Past, present and future"*. In L. Aceto, Z. Ésik, W.J. Fokkink, and A. Ingólfsdóttir, editors, Process Algebra: Open Problems and Future Directions, volume NS-03-3 of BRICS Notes Series, pages 7–12, 2003.
10. M. Abadi , C. Fournet , G. Gonthier, (1998) *"Secure Implementation of Channel Abstractions"*, Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science, p.105, June 21-24, 1998.
11. R. Milner. "*Polyadic π-Calculus : A Tutorial*". in F. L. Hamer, W. Brauer and H. Schwichtenberg, editors, Logic and Algebra of Specification. Springer-Verlag, 1993.
12. The UPAAL can be find in the web site: http://www.uppaal.com/
13. Fournet, C., G. Gonthier, J.-J. Lévy, L. Maranget, D. Rémy, "*A calculus of mobile agents*". Proc. 7th International Conference on Concurrency Theory (CONCUR'96), 406-421. 1996.
14. C. Fournet and G. Gonthier. "*The reflexive chemical abstract machine and the join-calculus*". In 23rd ACM Symposium on Principles of Programming Languages (POPL'96), 1996.
15. Robin Milner, Mads Tofte, Robert Harper, "*The Definition of Standard ML*", MIT Press 1990; (Revised edition adds author David MacQueen), MIT Press 1997.
16. http://jocaml.inria.fr/doc/index.html
17. Martin Odersky, "*Functional Nets*", Proc. European Symposium on Programming, Berlin, Germany, March 2000, pp. 1-25. Springer Lecture Notes in Computer Science 1782. Copyright © Springer Verlag.
18. G. Berry, and G. Boudol. "*The Chemical Abstract Machine*". Theoritical Computer Science, 96, 217-248, 1992.

Petri nets references:

19. Badouel, E., and Javier, O. "*Reconfigurable Nets, a Class of High Level Petri Nets Supporting Dynamic Changes within Workflow Systems*". Rapports de recherche - INRIA ISSN 0249-6399. 1998.
20. Andrea Asperti and Nadia Busi. "*Mobile Petri Nets*". Technical Report UBLCS-96-10, Department of Computer Science University of Bologna, May 1996. In the Mathematical Structures in Computer Science journal 19 (6): 1265-1278 (2009).
21. Valk, R. "*Petri Nets as Token Objects: An Introduction to Elementary Object Nets"*. In Applications and Theory of Petri Nets 1998, LNCS vol.1420, pp.1-25, Springer-Verlag, 1998.
22. Lomazova, I.A. *"Nested Petri Nets; Multi-level and Recursive Systems"*. Fundamenta Informaticae, vol.47, issue 3, pp. 283-293. IOS Press.
23. Bednarczyk, M.A., Bernardinello, L., Pawlowski, W., and Pomello, L. "*Modelling Mobility with Petri Hypernets*". In the 17th Int. Conf. on Recent Trends in Algebraic Development Techniques, WADT'04. LNCS vol. 3423, Springer-Verlag, 2004.
24. Rosa-Velardo, F. Marroqn Alonso, O. and Frutos Escrig, D. "*Mobile Synchronizing Petri Nets: a choreographic approach for coordination in Ubiquitous Systems*". In 1st Int. Workshop on Methods and Tools for Coordinating Concurrent, Distributed and Mobile Systems, MTCoord'05. ENTCS, No 150.
25. Dianxiang Xu and Yi Deng. "*Modeling Mobile Agent Systems with High Level Petri Nets*". In IEEE International Conference on Systems, Man, and Cybernetics, 2000. Volume: 5, page(s): 3177-3182.
26. Gilmore, S., Hillston, J., Kloul, L., and Ribaudo, M. "*PEPA nets: a structured performance modelling formalism*". In Performance Evaluation. Volume 54, Issue 2, October 2003, Pages 79-104. Elsevier.
27. Petri, C.A. "*KommuniKation mit Automaten*", Schriften des IIM Nr.2, Institut für Instrumentelle Mathematik, Bonn (1962). English translation: Technical Report RADC-TR-65-377, Griffiths Air Force Base, New York, vol. 1, suppl. 1 (1966).
28. Valk, R. "*Self Modifying Nets, A Natural Extension of Petri Nets"*. Proceeding of ICALP'78, Lecture Notes in Computer Science, vol. 62, pages 464-476. (1978).
29. Köhler, M. Moldt, D. and Rölke, H. "*Modelling mobility and mobile agents using nets within nets"*. In W. van der Aalst and E. Best, editors, Applications and Theory of Petri Nets 2003, Proceedings, volume 2679 of LNCS, pages 121–139. Springer-Verlag, 2003.
30. Valk, R. "*Object Petri nets: Using the nets-within-nets paradigm"*. Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, Advances in Petri Nets: Lectures on Concurrency and Petri Nets, volume 3098 of Lecture Notes in Computer Science, pages 819-848. Springer-Verlag, Berlin, Heidelberg, New York, 2004.
31. Jensen, K. "*An Introduction to the Theoretical Aspects of Coloured Petri Nets*". In J.W. de Bakker, W.-P. de Roever, G. Rozenberg (eds.), A Decade of Concurrency, Lecture Notes in Computer Science vol. 803, pp. 230-272. Springer-Verlag, Berlin/Heidelberg, 1994.
32. ZENIE (1985), "*Coloured stochastic Petri nets*", in: Proceedings of the International Workshop on Timed Petri Nets, Torino, IEEE Computer Society Press (262±271).
33. http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/quick.html. Many tools can be downloaded from this web site.
34. M. Buscemi and V. Sassone. "*High-Level Petri Nets as Type Theories in the Join Calculus*". In Proc. of Foundations of Software Science and Computation Structure (FoSSaCS '01), LNCS 2030, Springer-Verlag.

35. Cervesato, I.: "*Petri Nets and Linear Logic: a case study for logic programming*". In the Joint Conference on Declarative Programming, pp. 313–318, Italy, 11–14 September 1995.

36. Rosa-Velardo, F.: "*Coding Mobile Synchronizing Petri Nets into Rewriting Logic*". In Electronic Notes in Theoretical Computer science. Vol 174 , Issue 1, Pages 83-98. Elsevier, 2007.

Rewriting logics references:

37. Clavel, M. Durn, F. Eker, S. Lincoln, P. Mart-Oliet, N. Meseguer, J. and Quesada, J. "*Maude: specification and programming in rewriting logic*". SRI International, Januray 1999, http://maude.csl.sri.com.

38. Durán, F. Eker, S. Lincoln, P. and Meseguer, J. "*Principles of mobile maude"*. In D.Kotz and F.Mattern, editors, Agent systems, mobile agents and applications, second international symposium on agent systems and applications and fourth international symposium on mobile agents, ASA/MA 2000 LNCS 1882, Springer Verlag. Sept 2000.

39. Meseguer, J. "*Conditional rewriting logic as a unified model of concurrency*". Theoretical Computer Science, 96 (1):73-155, 1992.

40. Yasuyuki Tahara, Akihiko Ohsuga, Shinichi Honiden, "*Pigeon: a Specification Language for Mobile Agent Applications*", AAMAS'04, July 19-23, 2004, New York, USA.

41. Manuel Clavel, José Meseguer, Menlo Park, "*Reflection in conditional rewriting logic*". Journal of Theoretical Computer Science - Rewriting logic and its applications archive Volume 285 Issue 2, 28 August 2002. Elsevier Science Publishers Ltd. Essex, UK.

Our own References:

42. Kahloul, L. and Chaoui, A. '*Labeled Reconfigurable Nets for Modeling Code Mobility*'. In the International Arab Conference on Information Technology (ACIT'2007), November 26-28, Syria. 2007.

43. Kahloul, L. and Chaoui, A. '*Temporal Labeled Reconfigurable Nets for Code Mobility Modeling'*. Accepted for the International Workshop on Trustworthy Ubiquitous Computing (TwUC 2007) associated to the iiWAS2007 conference, Jakarta Indonesia on 3-5 December 2007.

44. Kahloul, L. and Chaoui, A. '*Code mobility modeling: a temporal labelled reconfigurable nets*'. In the Proceedings of the 1st International Conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications, MOBILWARE 2008, Innsbruck, Austria, February 13 - 15, 2008. ACM International Conference Proceeding Series 278 2008, ISBN 978-1-59593-984-5.

45. Kahloul, L. and Chaoui, A. '*Coloured Reconfigurable Nets for Code Mobility Modeling*'. In the International Journal of Computers, Communications & Control, ISSN 1841-9836, E-ISSN 1841-9844. Vol. III (2008), Suppl. issue: Proceedings of ICCCC 2008. pp 358-363.

46. Kahloul, L. and Chaoui, A. '*LRN/R-maude based approach for modeling and simulation of mobile code systems*'. In Ubiquitous Computing and Communication Journal (UbiCC journal), Volume 3 Number 6, Volume 3 No. 6, 12/20/2008. http://www.ubicc.org/search_advanced.aspx.

47. Kahloul, L., Chaoui, A. and Djouani, K. '*Modeling and Analysis of Reconfigurable Systems Using Flexible Nets'*. In the Second International Conference on Networked Digital Technologies, NDT 2010, Prague, Czech Republic, July 7-9, 2010. Proceedings, Part II. Volume 87. Published in the **"**Communications in Computer and Information Science**"** (CCIS) Series of Springer LNCS. Springer Berlin Heidelberg.

48. Kahloul, L, Chaoui, A and Djouani, K. "*Code Mobility Modelling: A Formal Study*". In International Review on Computer and Software, may 2009. http://www.praiseworthyprize.com/IRECOS.htm.

49. Kahloul Laid, Chaoui Allaoua and Djouani Karim, "*Modeling Reconfirgurable Systems Using Flexible Petri Nets"*, In 4th IEEE International Symposium on Theoretical Aspects of Software Engineering, August 24 - 27, 2010, Taipei, Taiwan.

Mobile computing references:

50. Agrawal, D. P. and Zeng, Q.A. '*Introduction to Wireless and Mobile Systems*', CL-Engineering Edition, August 2, 2002.

51. Fuggetta, A., Picco, G.P., and Vigna, G. *'Understanding Code Mobility'*. IEEE transactions on software engineering, vol. 24, no. 5, may 1998.

52. K.Arnold and J. Gosling. "The java specification language". Sun Micro System, 1996.

53. Andrea Barbu, "*Developing mobile agents through a formal*", Thesis prepared in french-german co-tutelle, Verteidigt am 12 September 2005 vor der Pr ufungskommission.

54. Carzaniga, G.P. Picco, and G. Vigna, "*Designing Distributed Applications with Mobile Code Paradigms*", *Proc. 19th Conf. Software Eng. (ICSE'97)*, R. Taylor, ed., pp. 22–32, ACM Press, 1997.

55. Fred Douglis. « *Process migration in the Sprite operating system* ». Technical Report UCB/CSD 87/343, Computer Science Division, University of California, Berkeley, February 1987.

56. George H.Forman, John Zahorjan, "The Challenges of Mobile Computing". UW CSE Tech Report #93-11-03 from ftp.cs.washington.edu. An edited version accepted in IEEE Computer.

57. Franklin, M. and S. Zdonik, "*Data In Your Face: Push Technology in Perspective*", ACM SIGMOD International Conference on Management of Data, pp. 516-519. 1998

58. George H.Forman and John Zahorjan, "*The Challenges of Mobile Computing*". This paper appears in: Computer. Issue Date: Apr 1994. Volume: 27. Issue: 4. On page(s): 38 - 47. IEEE computer society.

59. C. Ghezzi and G. Vigna, "*Mobile Code Paradigms and Technologies: A Case Study*", Rothermel and Popescu-Zeletin [72], pp. 39–49.

60. Robert Gray, David Kotz, Saurab Nog, Daniela Rus, George Cybenko, "*Mobile agents for mobile computing*", Technical Report PCS-TR96-285, 1996, ftp://ftp.cs.dartmouth.edu/TR/TR96-285.pdf

61. Carl Hewitt. "*The Apiary network architecture for Knowledgeable systems*". In Conference Record of the 1980 Lisp Conference, page 107-118, Palo Alto, California, August 1980. Stanford University.

62. Li Jingyue , "*Code Mobility Overview* ", (Essay for DIF 8914). Department of computer and information science, Norwegian University of Science and Technology, 2004.

63. Dag johansen, "*Mobile Agents: Right Concept, Wrong Approach*", Proceedings of the 2004 IEEE International Conference on Mobile Data Management (MDM'04)

64. E. Jul, H. Levy, N. Hutchinson, and A. Black, "*Fine-Grained Mobility in the Emerald System*," ACM Trans. Computer Systems, vol. 6, no. 2, pp. 109–133, Feb. 1988.

65. David Kotz and Robert S. Gray, "*Mobile Agents and the Future of the Internet*". Department of Computer Science/ Thayer School of Engineering Dartmouth College. In ACM Operating Systems Review, August 1999, pp. 7-13.

66. Danny B. Lange. "*Mobile Objects and Mobile Agents: The Future of Distributed Computing*". E. Jul (Ed.): ECOOP'98, LNCS 1445, pp.1 -12, 1998. Springer-Verlag Berlin Heidelberg 1998. This paper is based on a chapter of a book by Lange and Oshima entitled Programming and Deploying Java™ Mobile Agents with Aglets™, Addison-Wesley, 1998. (ISBN: 0-201-32582-9).

67. Danny B. Lange and Yariv Aridor. "*Agent Transfer Protocol – ATP/0.1*". IBM Corporation, 19 mars 1997.

68. Danny B. Lange and Mitsuru Oshima. "*Programming Mobile Agent in Java, With the Java Aglet API*". IBM Research, 1997. (Alpha5) Draft. IBM Corporation, 10 septembre 1997.

69. R. Lea, C. Jacquemont, and E. Pillevesse, "*COOL: System Support for Distributed Object-Oriented Programming*", Comm. ACM, vol. 36, no. 9, pp. 37–46, Nov. 1993.

70. X. Leroy. Objective Caml. http://pauillac.inria.fr/caml/, 1997

71. Edward D.Lazowska, Henry M.levy, Guy T. Almes, Michael J.Fisher, Robert J.Flower, and Stepher C.Vestal. "*The architecture of Eden Syetm*", In Proceeding of the 8th Symposium on Operating Systems Principles, pages 148-159, December 1981.

72. D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Os-hima, C. Tham, S. Virdhagriswaran, and J. White. "*MASIF: The OMG Mobile Agent System Interoperability Facility*". In Proceedings of the Second International Workshop on Mobile Agents, volume 1477 of Lecture Notes in Computer Science, pages 50-67, Stuttgart, Germany, September 1998. Springer-Verlag.

73. Nils P.Sudmann & Dag Johansen, « *software deployment using mobile agents* », In Proceeding CD '02 Proceedings of the IFIP/ACM Working Conference on Component Deployment Springer-Verlag London, UK 2002.

74. Saurab Nog, Sumit Chawla, and David Kotz."*An RPC mechanism for transportable agent*". Technical Report PCS-TR96-280, Dept. of Computer Science, Dartmouth College, March 1996.

75. Michael L.Powell and Barion P.Miller. « *Process migration in DEMOS/MP* ». In Proceeding of the Ninth ACM Symposium on Operating Systems Principles, pages 110-119, ACM/SIGOPS, October 1983.

76. Richard F.Rashid and George G.Roberison. « *Accent: A communication oriented network operating system kernel* ». In Proceeding of th 8 Symposium on Operating System Principles, pages 64-75, December 1981.

77. M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, P. Leonard, S. Langlois, and W. Neuhauser, "*Chorus Distributed Operating Systems*", Computing Systems, vol. 1, pp. 305–379, Oct. 1988.

78. Marvin M.Theimer, Keith A.Lantz and David R.Cheriton. « *Preemptable remote execution facilities for V-system* ». In proceedings of the 10th ACM Symposium on Operating Systems Principles, pages 2-12.ACM/SIGOPS, December 1985.

79. G. Thiel, "*Locus Operating System, A Transparent System*", Elsevier Computer Comm., vol. 14, no. 6, pp. 336–346, 1991.

80. Tommy Thorn, "*Programming languages for mobile code*", in the ACM Computing Surveys, 29(3): 213-239, Sept., 1997.

81. Bent Thomsen, Lone Leth, Frederick Knabe, and Pierre-Yves Chevalier. (1995) "*Mobile Agents*". ECRC external report, European Computer-Industry Research Center, 1995.

82. G. Vigna. (2004) "*Mobile agents: Ten reasons for failure*". In Proceedings of MDM Berkeley, CA, pages 298–299, January 2004.

83. J.E.White, (1994) "*Telescript technology: the foundation for the electronic market-place*", General magic Inc, Mountain View, CA, 1994.

84. J.E.White. (1994) "*Mobile Agent Make a Network an Open Platform for Third-Party Devlopers*". In Computer, 27(11): 89-90, November 1994. IEEE computer society.

85. Robert S. Gray. "*Agent Tcl: A flexible an secure mobile agent system*"; Thesis, Darmouth College, New Hamshir, USA, 30 juin 1997.

86. J.K. Boggs, "*IBM Remote Job Entry Facility: Generalize Subsystem Remote Job Entry Facility*," IBM Technical Disclosure Bulletin 752, IBM, Aug. 1973.

87. B. Gray. "*Soldiers, agents and wireless networks: A report on the actcomm scenarios and testbed*". In Proceedings of the 2000 Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology, 2000.

88. R. Pascotto. "*AMASE: A Complete Agent Platform for the Wireless Mobile Communication Environment*". T-Nova Deutsche Telekom Innovationsgesellschaft mbH Berkom,http://www.cordis.lu/infowin/acts/analysys/products/thematic/agents/ch3/amase.htm, 1998.

89. D. E. Brake, G. Emami, and GLOBAL INFOTEK VIENNA VA. "*Control of agent based systems (coabs) grid*". Technical Report A522524, Storming Media, http://www.stormingmedia.us/52/5225/A522524.html, June 2004.

90. H. Christensen. "*Cognitive (vision) systems*". In ERCIM News, Special: Cognitive Systems, 2003.

91. Shafi, U. Farooq, S. L. Kiani, M. Riaz, A. Shehzad, A. Ali, I. Legrand, and H. B. Newman. "*Diamonds - distributed agents for mobile & dynamic services*". CoRR, cs.DC/0305062, 2003.

92. M. Dam. "*Digital design and life-cycle management for distributed information supply services in innovation exploitation and technology transfer*". Technical Report IST-1999-10092, Hellenic TeleCommunications and Telematics Applications Company, 2001.

93. W. Thielmann and K. Rothermel. "Hawk: Harvesting the widely distributed knowledge". The web site of this project is: http://www.ipvs.uni-stuttgart.de/abteilungen/vs/forschung/projekte/HArvesting_the_Widely_Distributed_Knowledge/en.

94. B. Bauer, D. Bonnefoy, F. Bergenti, and R. Evans. "*The lightweight extensible agent platform*". In Proceedings of the Autonomous Agent Conference, February 2001.

95. O. Gutknecht and J. Ferber. "*Madkit: a generic multi-agent platform*". In Agents, pages 78–79, 2000.

96. T. Mota, S. Gouveris, G. Pavlou, A. Michalas, and J. Psoroulas. "*Quality of service management in ip networks using mobile agent technology*". In Proceedings of the IEEE/ACM International Workshop on Mobile Agents for Telecommunication Applications (MATA'2002), October 2002.

97. M. Weiss, C. Busch, and W. Schrter. "*Multimedia Arbeitsplatz der Zukunft - Assistenz und Delegation mit mobilen Softwareagenten*". Talheimer Verlag, 2003.

98. V. Vasudevan and S. Landis. "*Malleable services*". In Proceedings of the 34th Hawaii international Conference on System Sciences, October 2001. SysteMATech. http://www.systematech.org/index.php?Project.

99. SysteMATech. http://www.systematech.org/index.php?Project.

100. L. M. Camarihna-Matos and H. Afsarmanesh. "*telecare: Collaborative virtual elderly support communities*". In Proceedings of the 1st Workshop on Tele-Care and Collaborative Virtual Communities in Elderly Care, TELECARE, ISBN: 972-8865-10-4. ICEIS 2004, 2004.

101. "*PostScript language reference*", third edition,. Addison-Wesley Publishing Company. First printing February 1999. Can be downloaded from: http://www.adobe.com/products/postscript/pdfs/PLRM.pdf.

102. http://www.graphviz.org/Download.php. The graphViz tool can be downloaded freely from this site.